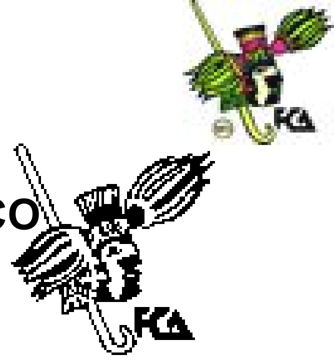


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

AUTOR: DAVID KANAGUSICO HERNÁNDEZ

Introducción a la programación		Clave: 1167
Plan: 2005		Créditos: 8
Licenciatura: Informática		Semestre: 1°
Área: Informática (Desarrollo de sistemas)		Hrs. asesoría: 4
Requisitos: Ninguno		Hrs. por semana: 4
Tipo de asignatura:	Obligatoria (x)	Optativa ( )

Objetivo general de la asignatura

Al finalizar el curso, el alumno será capaz de implantar algoritmos en un lenguaje de programación.

Temario oficial (horas sugeridas: 64)

1.	Introducción a la programación	4 hrs.
2.	Datos, constantes, variables, tipos, expresiones y asignaciones	6 hrs.
3.	Control de flujo	12 hrs.
4.	Funciones	14 hrs.
5.	Arreglos y estructuras	10 hrs.
6.	Manejo de apuntadores	8 hrs.
7.	Archivos	10 hrs.

Introducción

Los apuntes explican los puntos necesarios para el desarrollo de programas de computadora. Se tratan conceptos básicos y de la estructura de un programa, además de temas avanzados como son el uso de apuntadores y archivos.

En el *primer tema* (introducción a la programación) se mencionan conceptos básicos de programación. En el *segundo tema* (Datos, constantes, variables, tipos, expresiones y asignaciones) se enumeran dichos conceptos, los cuales son los elementos que construyen un programa. En el *tercer tema* (Control de flujo) se analiza la utilización de la estructura secuencial, condicional y repetitiva. En el *cuarto tema* (Funciones) se analiza la función, y su utilidad para la realización de tareas específicas dentro de un programa. En el *quinto tema* (Arreglos y estructuras) se desarrollan programas que utilizan los arreglos y estructuras para almacenar y manipular datos de un tipo, o de tipos diferentes. En el sexto tema (Manejo de apuntadores) se utilizan los apuntadores para la realización de programas que utilizan memoria dinámica. Por ultimo en el *séptimo tema* (Archivos) se expone el uso de los archivos y su utilidad para el almacenamiento de datos en la memoria secundaria de una computadora.

Tema 1. Introducción a la programación

Objetivo particular

El alumno conocerá los fundamentos de la programación, utilizando el paradigma de la programación estructurada.

Temario detallado

1.1. Concepto de lenguaje de programación
1.2. Programación estructurada
  1.2.1. Estructura de un programa en C
1.3. Programación orientada a objetos
1.4. Lenguaje máquina
1.5. Lenguaje de bajo nivel
1.6. Lenguajes de alto nivel
1.7. Intérpretes
1.8. Compiladores
1.9. Fases de compilación

Introducción

En este tema se desarrollaran los conceptos básicos de la programación, entendida ésta como la implemetación de un algoritmo (serie de pasos para resolver un problema) en un lenguaje de programación, dando como resultado un programa.

Un ejemplo es el sistema operativo Linux que fue desarrollado en el lenguaje C.

1.1. Concepto de lenguaje de programación

Un lenguaje de programación, es una herramienta que permite desarrollar programas para computadora.

La función de los lenguajes de programación es escribir programas que permiten la comunicación usuario/ máquina. Unos programas especiales (**compiladores o**

intérpretes) convierten las instrucciones escritas en código fuente, esto es, en instrucciones escritas en lenguaje máquina (0 y 1).

Para efectos del curso se utilizará el lenguaje C. Este lenguaje está caracterizado por ser de **uso general, de sintaxis compacta y portable**.

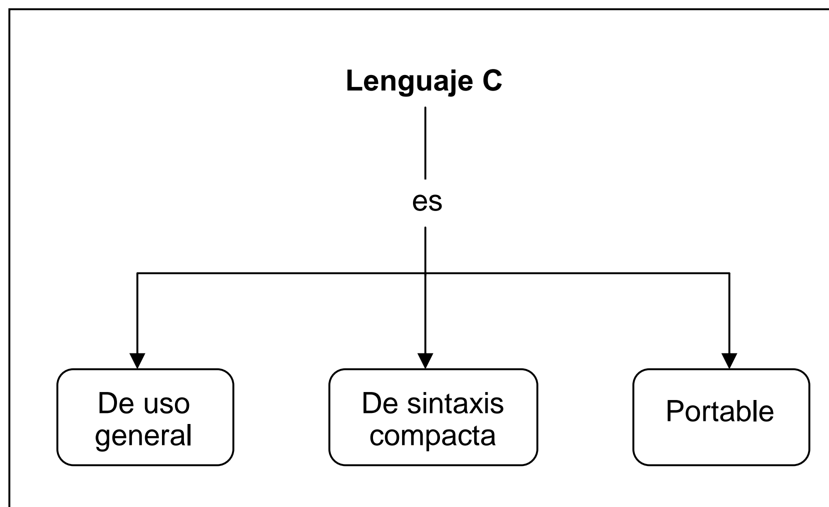


Figura 1.1. Características del lenguaje C

El lenguaje C es de uso general debido a que puede ser usado para desarrollar programas de diversa naturaleza, como lenguajes de programación, manejadores de bases de datos o sistemas operativos.

Su sintaxis es compacta, debido a que maneja pocas funciones y palabras reservadas, comparado con otros lenguajes como Java.
Además es portable, debido a que puede ser utilizado en varios sistemas operativos y hardware.

1.2. Programación estructurada

La programación estructurada es una forma de escribir programas.
La programación estructurada significa escribir un programa de acuerdo a las siguientes reglas:

- El programa tiene un **diseño modular**.
- Los **módulos** son diseñados de manera que un problema complejo, se divide en problemas más simples.
- Cada módulo se codifica utilizando las **tres estructuras** de **control básicas**: secuencia, selección y repetición.

C es un **lenguaje estructurado**. El componente estructural principal de C es la **función**, rutina que permite definir tareas de un programa y codificarlas por separado haciendo que los programas sean modulares.

1.2.1. Estructura de un programa en C

Todos los programas en C consisten en una o más funciones, la única función que siempre debe estar presente es la denominada **main()**, siendo la primera función que se invoca cuando comienza la ejecución del programa.

Forma general de un programa en C:

```
Archivos de cabecera
Declaraciones globales
tipo_devuelto main(parámetros)
{
      sentencias(s);
}

tipo_devuelto funcion(parámetros)
{
      sentencias(s);
}
```

Ejemplo de un programa en C.

Ejercicio 1.1.

```
# include <stdio.h>
main()
{
   printf("Hola mundo");
   return(0);
}
```

Este primer programa muestra un mensaje en pantalla. La primera línea que aparece se denomina *Encabezado* y es un archivo que proporciona información al compilador. En este archivo están incluidas las funciones **printf**.
La segunda línea del programa indica dónde comienza la función **main** indicando los valores que recibe y los que devuelve.

printf muestra un mensaje en la pantalla.

Por último, la sentencia **return(0)** indica que esta función no devuelve ningún valor.

El siguiente ejercicio permite introducir el mensaje que será mostrado en pantalla:

Ejercicio 1.2.

```
#include <stdio.h>

char s[100];
void main(void)
{
      printf("Introduzca el mensaje que desea desplegar\n");
      scanf("%s",s);
```

```
    printf("La cadena es: %s\n",s);
}
```

1.3. Programación orientada a objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación que define los programas en términos de “clases” de “objetos”, a éstos se les considera entidades que combinan **estado** (es decir, datos), **comportamiento** (procedimientos o métodos) e **identidad** (propiedad del objeto que lo diferencia del resto). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas.

1.4. Lenguaje máquina

Lenguaje de máquina es el **sistema de códigos directamente interpretable** por un circuito microprogramable, como el microprocesador de una computadora. Este lenguaje está compuesto por un conjunto de instrucciones que determinan acciones que serán por la máquina. Un programa de computadora consiste en una cadena de estas instrucciones de lenguaje de máquina (más los datos). Estas instrucciones son normalmente ejecutadas en secuencia, con eventuales cambios de flujo causados por el propio programa o eventos externos. El lenguaje de máquina es específico de cada máquina o arquitectura de la máquina, aunque el conjunto de instrucciones disponibles pueda ser similar entre ellas.

1.5. Lenguajes de bajo nivel

Un lenguaje de programación de bajo nivel es el que proporciona poca o ninguna abstracción del microprocesador de una computadora. Consecuentemente es fácil su trasladado al lenguaje máquina.

El término ensamblador (del inglés *assembler*) se refiere a un tipo de programa informático que se encarga de traducir un archivo fuente escrito en un lenguaje ensamblador, a un archivo objeto que contiene código máquina, ejecutable directamente por la máquina para la que se ha generado.

1.6. Lenguajes de alto nivel

Los lenguajes de Alto Nivel se caracterizan por expresar los algoritmos de una manera adecuada a la capacidad cognitiva humana, en lugar de a la capacidad ejecutora de las máquinas.

Ejemplos de lenguajes de alto nivel

- C++
- Fortran
- Java
- Perl
- PHP
- Python

C++ es un lenguaje de programación, diseñado a mediados de los años 1980, por Bjarne Stroustrup. Por otro lado C++ es un lenguaje que abarca dos paradigmas de la programación: la programación estructurada y la programación orientada a objetos.

Fortran es un lenguaje de programación desarrollado en los años 50 y activamente utilizado desde entonces. Acrónimo de "Formula Translator". Fortran se utiliza principalmente en aplicaciones científicas y análisis númerico.

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 1990. Las aplicaciones java están típicamente compiladas en un bytecode, aunque la compilación en código máquina nativo también es posible.

Perl, Lenguaje Práctico para la Extracción e Informe es un lenguaje de programación diseñado por Larry Wall creado en 1987. Perl toma características del C, del lenguaje interpretado shell sh, AWK, sed, Lisp y, en un grado inferior, muchos otros lenguajes de programación.

PHP es un lenguaje de programación usado frecuentemente para la creación de contenido para sitios web con los cuales se puede programar las paginas html y los códigos de fuente. PHP es un acrónimo que significa "PHP Hypertext Pre-processor" (inicialmente PHP Tools, o, Personal Home Page Tools), y se trata de

un lenguaje interpretado usado para la creación de aplicaciones para servidores, o creación de contenido dinámico para sitios web. Últimamente también para la creación de otro tipo de programas incluyendo aplicaciones con interfaz gráfica usando las librerías Qt o GTK+.

Python es un lenguaje de programación creado por Guido van Rossum en el año 1990. En la actualidad Python se desarrolla como un proyecto de código abierto, administrado por la Python Software Foundation. La última versión estable del lenguaje es actualmente (septiembre 2006) la 2.5 .

1.7. Intérpretes

Un intérprete es un programa que analiza y ejecuta un código fuente, toma un código, lo traduce y a continuación lo ejecuta.

Como ejemplo de lenguajes interpretados tenemos a: PHP, Perl y Python.

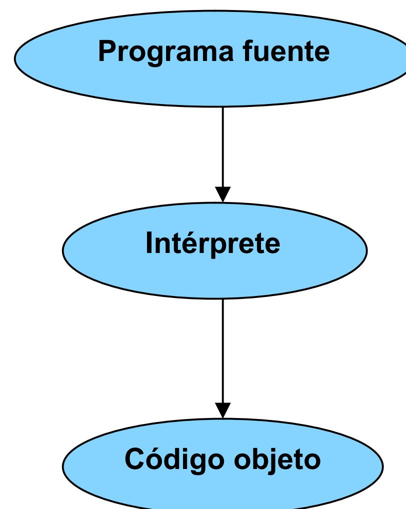


Figura 1.2. Funcionamiento de un intérprete

1.8. Compiladores

Un compilador es un programa (o un conjunto de programas) que traduce un programa escrito en código fuente, generando un programa en código objeto. Este proceso de traducción se conoce como **compilación**.

Posterior a esto, al código objeto se le agregan las librerías a través de un programa llamado **linker**, y se obtiene el código ejecutable.

Ejemplo de lenguajes que utiliza un compilador tenemos a **C, C++, Visual Basic.**

Las notas harán referencia al lenguaje C, en este lenguaje se realizaran los ejemplos. El compilador de C lee el programa y lo convierte a código objeto. Una vez compilado, las líneas de código fuente dejan de tener sentido. Este código objeto puede ser ejecutado por la computadora. El compilador de C incorpora una biblioteca estándar que proporciona las funciones necesarias para llevar a cabo las tareas más usuales.

1.9. Fases de la compilación

La compilación permite crear un programa de computadora que puede ser ejecutado por una computadora.

La compilación de un programa consiste en tres pasos.

1. Creación del código fuente,
2. Compilación del programa y
3. Enlace del programa con las funciones necesarias de la biblioteca.

La forma en que se lleve a cabo el enlace variará entre distintos compiladores, pero la forma general es:

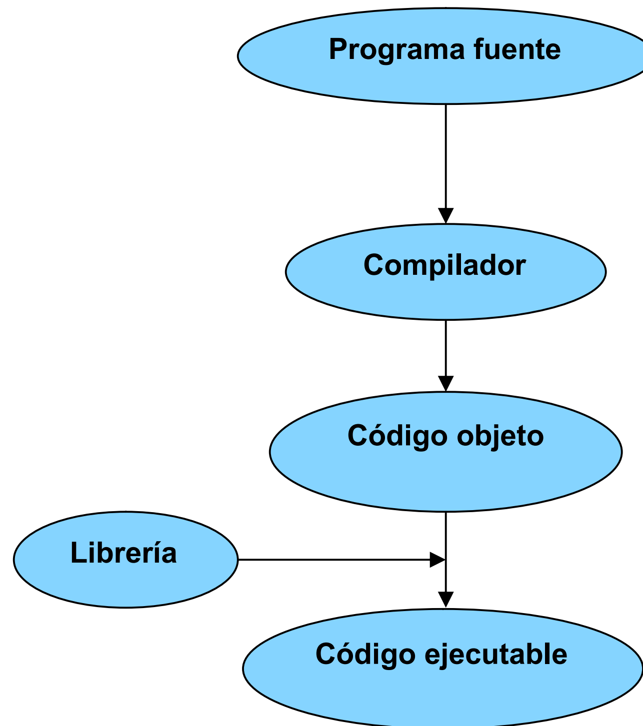


Figura 1.3. Proceso de Compilación

CONCLUSIONES

Los lenguajes de programación utilizan diversos paradigmas y dependiendo del problema que se desee resolver, se debe escoger el más adecuado. Por ejemplo los lenguajes orientados a objetos permiten un desarrollo ágil de los programas, además los compiladores e intérpretes, permiten la depuración y ejecución de un programa de computadora.

Fuentes web del tema 1

JOYANES, Luis. "*Fundamentos de Programación",* España: Pearson Prentice Hall, 3ª Edición, 2003, 1004 pp.

www.wikipedia.org

Actividades de aprendizaje

A.1.1. Representa en un diagrama de flujo el funcionamiento de un intérprete.

A.1.2. Investiga 5 lenguajes de programación que utilicen un intérprete y 5 que utilicen un compilador.

A.1.3. Investiga 5 lenguajes de programación que sean “libres” y elabora un cuadro comparativo.

A.1.4. Investiga el significado de “librería” y “linker”

A.1.5. Investiga que significa el término “IDE”.

Cuestionario de autoevaluación

1. Señale qué es la programación estructurada.
2. Mencione qué es la programación orientada a objetos.
3. ¿Qué es un compilador?
4. ¿Qué es un intérprete?
5. Defina qué es un *linker*.
6. Describa qué es el código fuente.
7. ¿Qué es el código objeto?
8. en que consiste el código ejecutable.
9. ¿Qué es un lenguaje de bajo nivel?
10. ¿Qué es un lenguaje de alto nivel?

Examen de autoevaluación

1. Un lenguaje de programación es:
    a. Una serie de pasos para resolver un problema.
    b. Un diagrama de flujo.
    c. Un paradigma.
    d. Conjunto de normas que permiten escribir un programa, para su procesamiento por una computadora.

2. La programación estructurada:
    a. Es un lenguaje de programación.
    b. Divide un problema complejo en problemas sencillos.
    c. Divide un programa en objetos.
    d. Solo utiliza intérpretes.

3. La programación heurística proporciona:
    a. Métodos de aprendizaje.
    b. Una solución a un problema.
    c. Opciones de solución a un problema.
    d. La solución óptima.

4. PROLOG utiliza:
    a. listas.
    b. axiomas y teoremas.
    c. funciones.
    d. sentencias.

5. La programación estructurada utiliza:
    a. Selección.
    b. Repetición.
    c. Iteración.
    d. Selección, Repetición e Iteración.

6. Es un programa que trasforma el código fuente en código ejecutable:
    a. Interprete
    b. Lenguaje de programación.
    c. Compilador.
    d. Librería.

7. Es un ejemplo de lenguaje de alto nivel:
    a. PHP
    b. Ensamblador.
    c. Lenguaje máquina.
    d. Lenguaje C.

8. Es un ejemplo de lenguaje de bajo nivel:
    a. C.
    b. Ensamblador.
    c. C++.
    d. Java.

9. Es un ejemplo de lenguaje que utiliza un intérprete:
    a. C.
    b. Python.
    c. C++.
    d. Fortran.

10. Es un programa que agrega librerías:
    a. Compilador
    b. Intérprete.
    c. Linker.
    d. Lenguaje de programación.

Tema 2. Tipos de datos, constantes, variables, expresiones y estructura de un programa

Objetivo particular

El alumno aprenderá los conceptos de tipo de dato, constante, variable y expresión, además se analizará la estructura de un programa informático.

Temario detallado

2.1. Tipos de datos
- 2.1.1. Tipos de datos elementales
- 2.1.2. Tipos definidos por el usuario

2.2. Palabras reservadas

2.3. Identificadores

2.4. Operadores
- 2.4.1. Asignación
- 2.4.2. Operadores aritméticos

2.5. Expresiones y reglas de prioridad
- 2.5.1. Prioridad de los operadores aritméticos

2.6. Variables y constantes
- 2.6.1. Constantes
- 2.6.2. Variables
- 2.6.3. Conversión
- 2.6.4. Ámbito de variables

2.7. Estructura de un programa

Introducción

Un tipo de dato lo podemos definir a partir de sus valores permitidos (entero, carácter, etc), por otro lado las palabras reservadas, son utilizadas por un lenguaje de programación para fines específicos, y no pueden ser utilizadas por el programador, los identificadores se utilizan para diferenciar variables y constantes en un programa. Las variables permiten que un identificador pueda tomar varios

valores, por el contrario las constantes son valores definidos que no pueden cambiar durante la ejecución del programa. Las expresiones se forman combinando constantes, variables y operadores, todos estos elementos permiten la creación de un programa informático.

2.1. Tipos de datos

2.1.1. Tipos de datos elementales

Las formas de organizar datos están determinadas por los **tipos de datos** definidos en el lenguaje.

Un tipo de dato determina el **rango de valores** que puede tomar el objeto, las **operaciones** a que puede ser sometido y el **formato** de almacenamiento en memoria.

En C:

- Existen tipos predefinidos
- El usuario puede definir otros tipos, a partir de los básicos.

Esta tabla describe los tipos de datos que se pueden utilizar en el lenguaje C.

TIPO	RANGO DE VALORES	TAMAÑO EN BYTES	DESCRIPCIÓN
char	-128 a 127	**1**	Para una letra o un dígito.
unsigned char	0 a 255	**1**	Letra o número positivo.
Int	-32.768 a 32.767	**2**	Para números enteros.
unsigned int	0 a 65.535	**2**	Para números enteros.
long int	±2.147.483.647	**4**	Para números enteros
unsigned long int	0 a 4.294.967.295	**4**	Para números enteros
float	3.4E-38 decimales(6)	**6**	Para números con decimales
double	1.7E-308 decimales(10)	**8**	Para números con decimales
long double	3.4E-4932	**10**	Para números con

	decimales(10)		decimales

Ejemplo de utilización de tipos de datos en C.

```
int numero;
int numero = 10;
float numero = 10.23;
```

Veamos un programa que determina el tamaño en bytes de los tipos de datos fundamentales en C.

Ejercicio 2.1.

```
# include <stdio.h>
void main()
{
     char c;
     short s;
     int i;
     long l;
     float f;
     double d;
     printf ("Tipo char: %d bytes\n", sizeof(c));
     printf ("Tipo short: %d bytes\n", sizeof(s));
     printf ("Tipo int: %d bytes\n", sizeof(i));
     printf ("Tipo long: %d bytes\n", sizeof(l));
     printf ("Tipo float: %d bytes\n", sizeof(f));
     printf ("Tipo double: %d bytes\n", sizeof(d));
}
```

.

El resultado de este programa es el siguiente:

```
"C:\ejercicio1\Debug\ejercicio1.exe"
Tipo char: 1 bytes
Tipo short: 2 bytes
Tipo int: 4 bytes
Tipo long: 4 bytes
Tipo float: 4 bytes
Tipo double: 8 bytes
Press any key to continue_
```

2.1.2. Tipos definidos por el usuario

C permite nuevos nombres para tipos de datos. Realmente no se crea un nuevo tipo de dato, sino que se define uno nuevo para un tipo existente. Esto ayuda a hacer más transportables los programas que dependen de las máquinas, sólo habrá que cambiar las sentencias *typedef* cuando se compile en un nuevo entorno. Las sentencias *typedef* permiten la creación de nuevos tipos de datos.

SINTAXIS:

```
typedef tipo nuevo_nombre;
nuevo_nombre nombre_variable[=valor];
```

Por ejemplo, se puede crear un nuevo nombre para **float** usando:

```
typedef float balance;
```

2.2. Palabras reservadas

Las palabras reservadas son algunos símbolos cuyo significado está predefinido y no se pueden usar para otro fin, aquí tenemos algunas palabras reservadas en el lenguaje C.

auto	break	case	char	continue	default	do	double
else	enum	extern	float	for	goto	if	int
long	register	return	short	sizeof	static	struct	switch
typedef	union	unsigned	void	while			

2.3. Identificadores

Los identificadores tienen como característica lo siguiente:

- Son secuencias de caracteres que se pueden formar usando letras, cifras y el carácter de subrayado "_".
- Se usan para dar nombre a los objetos que se manejan en un programa: tipos, variables, funciones.
- Deben comenzar por letra o por "_".
- Se distingue entre mayúsculas y minúsculas.
- Se deben definir en sentencias de declaración antes de ser usados.

SINTAXIS:

```
int i, j, k;
float largo, ancho, alto;
enum colores {rojo, azul, verde}
color1, color2;
```

2.4. Operadores

C es un lenguaje muy rico en operadores incorporados, es decir, implementados al realizarse el compilador. Se pueden utilizar operadores: **aritméticos, relacionales**

y lógicos. También se definen operadores para realizar determinadas tareas, como las asignaciones.

2.4.1. Asignación

En C se utiliza signo de **=** como operador de asignación, al utilizarlo se realiza esta acción. El operador destino (parte izquierda) debe ser siempre una variable, mientras que en la parte derecha puede estar cualquier expresión válida.

SINTAXIS:

```
variable=valor;
variable=variable1;
variable=variable1=variable2=variableN=valor;
```

2.4.2. Operadores Aritméticos

Los operadores aritméticos pueden aplicarse a todo tipo de expresiones. Son utilizados para realizar operaciones matemáticas sencillas, aunque uniéndolos se puede realizar cualquier tipo de operaciones. En la siguiente tabla se muestran todos los operadores aritméticos.

OPERADOR	DESCRIPCIÓN
-	Resta.
+	Suma
*****	Multiplica
/	Divide
%	Módulo (resto de una división)
-	Signo negativo
--	Decremento en 1.
++	Incrementa en 1.

Tabla 2.1. Operadores aritméticos

- Corresponden a las operaciones matemáticas de suma, resta, multiplicación, división y módulo.
- Son binarios porque cada uno tiene dos operandos.
- Hay un operador unario menos "-", pero no hay operador unario más "+":

-3 es una expresión correcta

+3 No es una expresión correcta

La **división de enteros** devuelve el cociente entero y desecha la fracción restante:

1/2 tiene el valor 0

3/2 tiene el valor 1

-7/3 tiene el valor -2

El **operador módulo** se aplica así: con dos enteros positivos, devuelve el resto de la división.

12%3 tiene el valor 0

12%5 tiene el valor 2

Los **operadores de incremento y decremento** son unarios.

- Tienen la misma prioridad que el menos "-" unario.
- Se asocian de derecha a izquierda.
- Pueden aplicarse a variables, pero no a constantes ni a expresiones.
- Se pueden presentar como prefijo o como sufijo.
- Aplicados a variables enteras, su efecto es incrementar o decrementar el valor de la variable en una unidad:

++i; es equivalente a i=i+1;

--i; es equivalente a i=i-1;

Cuando se usan en una expresión, se produce un efecto secundario sobre la variable:

El valor de la variable se incrementa antes o después de ser usado.

con ++a el valor de "a" se incrementa antes de evaluar la expresión.
con a++ el valor de "a" se incrementa después de evaluar la expresión.
con a el valor de "a" no se modifica antes ni después de evaluar la expresión.

Ejemplos:
a=2*(++c), se incrementa el valor de "c" y se evalúa la expresión después.
Es equivalente a: c=c+1; a=2*c;

a[++i]=0 se incrementa el valor de "i" y se realiza la asignación después.
Es equivalente a: i=i+1; a[i]=0;

a[i++], se realiza la asignación con el valor actual de "i", y se incrementa el valor de "i" después.
Es equivalente a: a[i]=0; i=i+1;

2.4.3. Lógicos y relacionales

Los operadores relacionales hacen referencia a la relación entre unos valores y otros; los lógicos, a la forma en que esas relaciones pueden conectarse entre sí. Los veremos a la par por la estrecha relación en la que trabajan.

OPERADORES RELACIONALES	
OPERADOR	**DESCRIPCIÓN**
<	Menor que.
>	Mayor que.
<=	Menor o igual.
>=	Mayor o igual
= =	Igual
! =	Distinto

OPERADORES LÓGICOS	
OPERADOR	**DESCRIPCIÓN**
&&	Y (AND)
| |	O (OR)
!	NO (NOT)

Tabla 2.2. Operadores relacionales y lógicos

Ejercicio 2.2.

El siguiente ejercicio muestra el uso de los operadores lógicos y relacionales. El usuario debe introducir dos valores numéricos y el programa hace comparaciones relaciones y lógicas con los valores introducidos.

```
/*Este programa en C muestra el uso de los operadores
lógicos y relacionales */
# include <stdio.h>
main()
{
      float valor1, valor2;
      printf("Por favor introduzca valor 1: ");
      scanf("%f",&valor1);
      printf("Por favor introduzca valor 2: ");
      scanf("%f",&valor2);
      printf("\n");
      printf("valor1 > valor2 es %d\n", (valor1>valor2));
      printf("valor1 < valor2 es %d\n", (valor1<valor2));
```

```
        printf("valor1 >= valor2 es %d\n", (valor1>=valor2));
        printf("valor1 <= valor2 es %d\n", (valor1<=valor2));
        printf("valor1 == valor2 es %d\n", (valor1==valor2));
        printf("valor1 != valor2 es %d\n", (valor1!=valor2));
        printf("valor1 && valor2 es %d\n", (valor1&&valor2));
        printf("valor1 || valor2 es %d\n", (valor1||valor2));
        return(0);
}
```

.

El resultado de este programa es el siguiente:

```
"C:\Archivos de programa\Microsoft Visual Studio\MyProjects\ejemplo1\Debug\ejemplo1.exe"
Por favor introduzca valor 1: 1
Por favor introduzca valor 2: 2

valor1 > valor2 es 0
valor1 < valor2 es 1
valor1 >= valor2 es 0
valor1 <= valor2 es 1
valor1 == valor2 es 0
valor1 != valor2 es 1
valor1 && valor2 es 1
valor1 || valor2 es 1
Press any key to continue_
```

Veamos un programa que indica el menor de dos números leídos.

Ejercicio 2.3.

```
/* Indica el menor de dos enteros leídos */
#include <stdio.h>
void main ( )
{
        int n1, n2, menor (int, int);
```

```
        printf ("Introducir dos enteros:\n");
        scanf ("%d%d", &n1, &n2);
        if ( n1 == n2 )
              printf ("Son iguales \n");
        else
              printf ("El menor es: %d\n",menor(n1,n2));
}
int menor (int a, int b)
{
        if ( a < b )
              return (a );
        else
              return ( b );
}
```

El resultado del programa es el siguiente:

```
"C:\Archivos de programa\Microsoft Visual Studio\MyProjects\ejemplo1\Debug\ejemplo1.exe"
Introducir dos enteros:
1
2
El menor es: 1
Press any key to continue_
```

2.5. Expresiones y reglas de prioridad

Una expresión se forma combinando constantes, variables, operadores y llamadas a funciones.

Ejemplo:

s = s + i

n == 0

++i

Una **expresión** representa un valor, el resultado de realizar las operaciones indicadas siguiendo las reglas de evaluación establecidas en el lenguaje.

Con expresiones se forman **sentencias**; con éstas, **funciones**, y con estas últimas se construye un **programa completo**.

Cada expresión toma un valor que se determina tomando los valores de las variables y constantes implicadas y la ejecución de las operaciones indicadas.

Una expresión consta de operadores y operandos. Según sea el tipo de datos que manipulan, se clasifican las expresiones en:

- Aritméticas
- Relaciónales
- Lógicas

2.5.1. Prioridad de los operadores aritméticos

Son prioridades de los operadores matemáticos:

- Todas las expresiones entre paréntesis se evalúan primero. Las expresiones con paréntesis anidados se evalúan de dentro hacia afuera, el paréntesis más interno se evalúa primero.
- Dentro de una misma expresión los operadores se evalúan en el siguiente orden.

1. () Paréntesis
2. ^ Exponenciación
3. *, /, mod Multiplicación, división, módulo. (El módulo o mod es el resto de una división)
4. +, - Suma y resta.

Los operadores en una misma expresión con igual nivel de prioridad se evalúan de izquierda a derecha.

Ejemplos:

a. 4 + 2 * 5 = 14 — Primero se multiplica y después se suma
b. 23 * 2 / 5 = 9.2 — 46 / 5 = 9.2
c. 3 + 5 * (10 - (2 + 4)) = 23 — 3 + 5 * (10 - 6) = 3 + 5 * 4 = 3 + 20 = 23
d. 2.1 * (1.5 + 3.0 * 4.1) = 28.98 — 2.1 * (1.5 + 12.3) = 2.1 * 13.8 = 28.98

2.6. Variables y constantes

2.6.1. Constantes

Las constantes se refieren a los **valores fijos** que no pueden ser modificados por el programa. Las **constantes de carácter** van encerradas en comillas simples. Las **constantes enteras** se especifican con números sin parte decimal y las de **coma flotante** con su parte entera separada por un punto de su parte decimal.

Las constantes son entidades cuyo valor no se modifica durante la ejecución del programa.

Hay constantes de varios tipos.

Ejemplos:

numéricas: `-7 3.1416 -2.5e-3`

caracteres: `'a' '\n' '\0'`

cadenas: `"indice general"`

SINTAXIS:

```
const tipo nombre=valor_entero;
const tipo nombre=valor_entero.valor_decimal;
const tipo nombre='carácter';
```

Otra manera de usar constantes es a través de la directiva **#define, define** es un identificador y una secuencia de caracteres que se sustituirá cada vez que se encuentre éste en el archivo fuente. También pueden ser utilizados para definir valores numéricos constantes.

SINTAXIS:

```
#define IDENTIFICADOR valor_numerico
#define IDENTIFICADOR "cadena"
```

Ejercicio 6.1.

```
#include <stdio.h>
#include <conio.h>
#define DOLAR 11.50
#define TEXTO "Esto es una prueba"
const int peso=1;
void main(void)
{
      printf("El valor del Dolar es %.2f pesos",DOLAR);
      printf("\nEl valor del Peso es %d ",peso);
      printf("\n%s",TEXTO);
      printf("\nEjemplo de constantes y defines");
      getch();
```

```
}
```

.

El resultado del programa es el siguiente:

```
"C:\Archivos de programa\Microsoft Visual Studio\MyProjects\ejemplo1\Debug\ejemplo1.exe"
El valor del Dolar es 11.50 pesos
El valor del Peso es 1
Esto es una prueba
Ejemplo de constantes y defines_
```

2.6.2. Variables

Unidad básica de almacenamiento, la creación de una variable es la *combinación* de un *identificador,* un *tipo* y un *ámbito*. Todas las variables en C deben ser declaradas antes de ser usadas.

Las variables, también conocidas como *identificadores*, deben cumplir las siguientes reglas:

- La longitud puede ir de 1 carácter a 31.
- El primero de ellos debe ser siempre una letra.
- No puede contener espacios en blanco, ni acentos y caracteres gramaticales.
- Hay que tener en cuenta que el compilador distingue entre mayúsculas y minúsculas.

SINTAXIS:

```
tipo nombre=valor_numerico;
tipo nombre='letra';
tipo nombre[tamaño]="cadena de letras",
tipo nombre=valor_entero.valor_decimal;
```

2.6.3. Conversión

Las conversiones (*casting*) automáticas pueden ser controladas por el programador. Bastará con anteponer y encerrar entre paréntesis el tipo al que se desea convertir. Este tipo de conversiones sólo es temporal y la variable a convertir mantiene su valor.

SINTAXIS:

```
variable_destino=(tipo)variable_a_convertir;
variable_destino=(tipo)(variable1+variable2+variableN);
```

Ejemplo:

Convertimos 2 variables float para guardar la suma en un entero.

La biblioteca *conio.h* define varias funciones utilizadas en las llamadas a rutinas de entrada/salida por consola de dos.

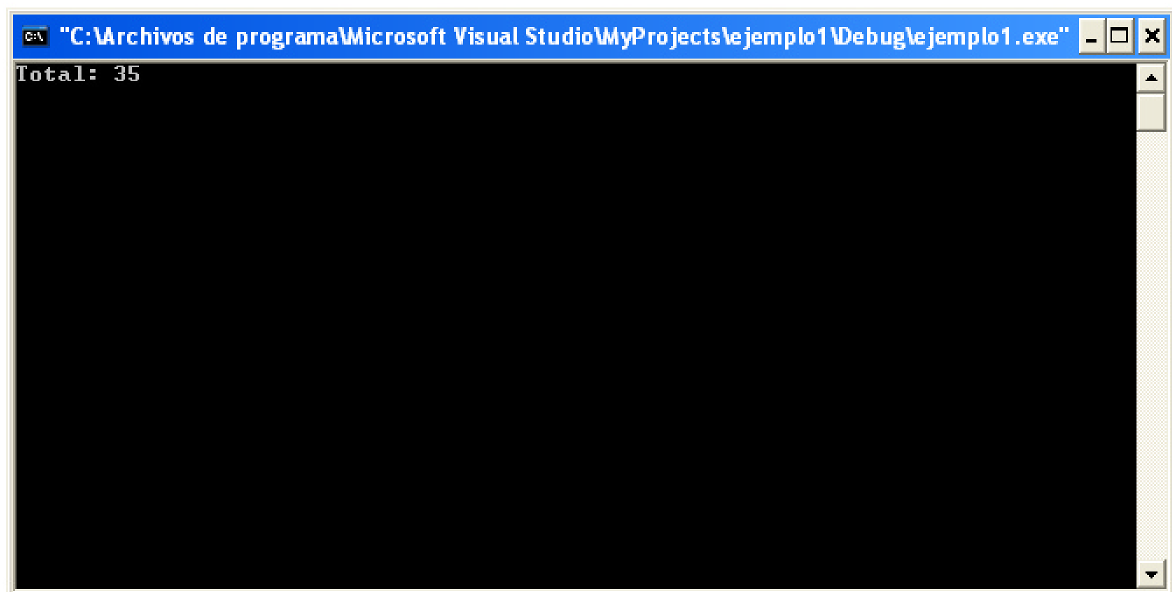
Ejercicio 6.1.

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
```

```
        float num1=25.75, num2=10.15;
        int total=0;

        total=(int)num1+(int)num2;
        printf("Total: %d",total);
        getch();
}
```

El resultado del programa es el siguiente:



2.6.4. Ámbito de variables

Según el lugar donde se declaren las variables tendrán un ámbito. Según el ámbito de las variables pueden ser utilizadas desde cualquier parte del programa o únicamente en la función donde han sido declaradas. Las variables pueden ser:

LOCALES: Cuando se declaran dentro de una función. Pueden ser referenciadas (utilizadas) por sentencias que estén dentro de la función que han sido declaradas. No son conocidas fuera de su función. Pierden su valor cuando se sale de la función.

GLOBALES: Son conocidas a lo largo de todo el programa, y se pueden usar desde cualquier parte del código. Mantienen sus valores durante toda la ejecución. Deben ser declaradas fuera de todas las funciones incluida main(). La sintaxis de creación no cambia nada con respecto a las variables locales.

DE REGISTRO: Otra posibilidad es, que en vez de ser mantenidas en posiciones de memoria de la computadora, se las guarde en registros internos del microprocesador. De esta manera el acceso a ellas es más directo y rápido. Para indicar al compilador que es una variable de registro hay que añadir a la declaración la palabra *register* delante del tipo. Solo se puede utilizar para variables locales.

ESTÁTICAS: Las variables locales nacen y mueren con cada llamada y finalización de una función, sería útil que mantuvieran su valor entre una llamada y otra sin por ello perder su ámbito. Para conseguir eso se añade a una variable local la palabra *static* delante del tipo.

EXTERNAS: Debido a que en C es normal la compilación por separado de pequeños módulos que componen un programa completo, puede darse el caso de que deba utilizar una variable global que se conozca en los módulos que nos interesen sin perder su valor. Añadiendo delante del tipo la palabra *extern* y definiéndola en los otros módulos como global ya tendremos nuestra variable global.[1]

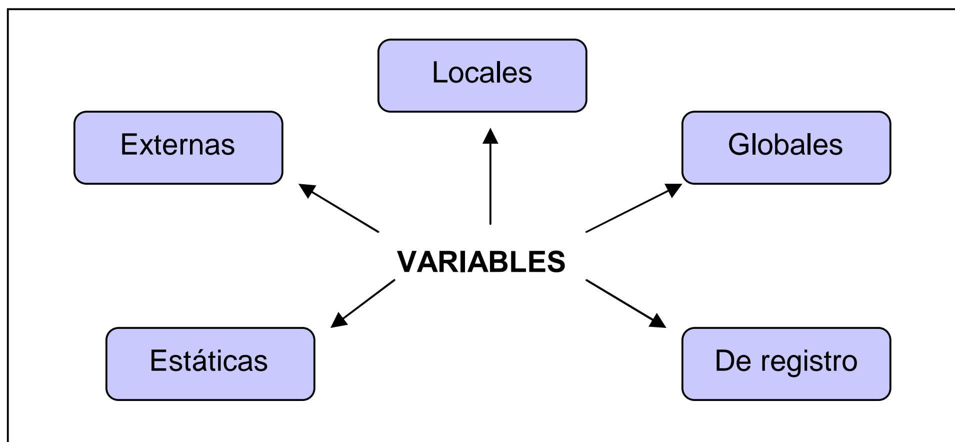


Figura 2.1. Tipos de variables

[1] Fuente: www.lawebdelprogramador.com, (20 de mayo del 2007.)

2.7. Estructura de un programa

Ya hemos visto varios programas en C, sin embargo no hemos revisado su estructura, por lo que analizaremos sus partes, con el último programa visto:

```
#include <stdio.h>          <-BIBLIOTECA
#include <conio.h>          <-BIBLIOTECA
```

Las bibliotecas son repositorios de funciones, la biblioteca **stdio.h** contiene funciones para la entrada y salida de datos, un ejemplo de esto es la función **printf**.

```
void main(void)          <-FUNCION PRINCIPAL
```

La función main es la función principal de todo programa en C.

```
{   LLAVES DE INICIO
```

Las llaves de inicio de fin indican cuando inicia y termina un programa

```
float num1=25.75, num2=10.15; <-DECLARACION DE VARIABLES
int total=0;                  <-DECLARACION DE VARIABLES
```

Es esta parte se declaran las variables que se van usar, su tipo, y en su caso se inicializan.

```
total=(int)num1+(int)num2;      <-DESARROLLO DEL PROGRAMA
printf("Total: %d",total);      <-DESARROLLO DEL PROGRAMA
getch();                        <-DESARROLLO DEL PROGRAMA
```

En esta parte como su nombre lo indica, se desarrolla el programa.

```
}        LLAVES DE FIN
```

Estas son las partes más importantes de un programa en C.

CONCLUSIONES

Los tipos de datos, las palabras reservadas y los identificadores permiten la creación de variables, constantes y funciones de un programa.

Los operadores permiten la manipulación de las variables.

Por último Las variables permiten cambiar el contenido a lo largo de la ejecución de un programa, lo contrario de las constantes que son datos permanentes, por último las expresiones permiten realizar operaciones aritméticas, relacionales y lógicas en un programa informático.

Bibliografía del tema 2

www.lawebdelprogramador.com

www.monografias.com

Actividades de aprendizaje

A.2.1. Elabora un programa que obtenga la calificación de un alumno, se promediarán 3 exámenes y 3 trabajos.

A.2.2. Diseña un programa que eleve al cubo un número introducido por el usuario.

A.2.3. Realiza un programa que obtenga el área de un triángulo, con datos introducidos por el usuario.

A.2.4. Realiza un programa que transforme metros a su equivalente en kilómetros, con datos introducidos por el usuario.

A.2.5. Realiza un programa que transforme a pesos una cantidad introducida en dólares.

Cuestionario de autoevaluación

1. Señale qué significa la palabra *auto*.
2. Indique en que consiste el *long*.
3. ¿Qué significa la palabra *sizeof*?
4. ¿Qué significa la palabra *short*?
5. Mencione el significado de la palabra *goto*.
6. ¿Qué significa la palabra *register*?.
7. Mencione qué significa la palabra *extern*.
8. ¿Qué significa la palabra *static*?
9. ¿Qué significa la palabra *continue*?
10. Describa el significado de la palabra *unsigned*.

Examen de autoevaluación

1. Un tipo de dato:
   a. Permite usar varios elementos en una estructura.
   b. Determina las clases a usar.
   c. Determina los objetos a usar.
   d. Permite usar un rango de datos.

2. La siguiente es una palabra reservada en C:
   a. Si
   b. Entonces.
   c. If.
   d. Mientras.

3. Un ejemplo de un operador relacional es:
   a. >
   b. AND
   c. &&

d. %

4. El operador modulo obtiene:
   a. El incremento de una variable.
   b. El decremento de una variable.
   c. La división de una variable.
   d. El resto de una división.

5. Las variables locales se declaran:
   a. Dentro de una función
   b. Después de main()
   c. Fuera de una función
   d. Antes de main()

6. El nombre de una variable no debe sobrepasar los:
   a. 20 caracteres.
   b. 30 caracteres.
   c. 31 caracteres.
   d. 32 caracteres.

7. El operador aritmético de mayor precedencia es el:
   a. -
   b. +
   c. /
   d. ( )

8. Son variables que se declaran dentro de una función:
   a. Locales.
   b. Globales.
   c. Estáticas.
   d. Externas.

9. Para definir nuevos tipos de datos se utiliza la palabra reservada:
    a. Register.
    b. Struct.
    c. Casting.
    d. Typedef.

10. Palabra reservada para definir constantes:
    a. #include
    b. #define
    c. stdio.h
    d. return()

TEMA 3. CONTROL DE FLUJO

Objetivo particular

Comprender y utilizar las tres estructuras principales de la programación estructurada, que son la estructura secuencial, la estructura alternativa y la estructura repetitiva.

Temario detallado

3.1. Estructura secuencial

3.2. Estructura alternativa

3.3. Estructura repetitiva

Introducción

La programación estructurada nos dice que es posible crear cualquier tipo de programa de computadora utilizando tres estructuras: la secuencial, la alternativa y la repetitiva. En el tema de control de flujo examinaremos la forma en que trabajan estas estructuras, además de ejemplos de su utilización.

3.1. Estructura secuencial

El control de flujo se refiere al orden en que se ejecutan las sentencias del programa. A menos que se especifique expresamente, el flujo normal de control de todos los programas es secuencial.

La estructura secuencial ejecuta las acciones sucesivamente,. sin posibilidad de omitir ninguna y sin bifurcaciones. Todas estas estructuras tendrán una entrada y una salida.

Inicio

Acción a_1

Acción a_2

.

.

Acción a_n

Fin

Veamos algunos programas que utilizan la estructura secuencial.

Ejercicio 3.1.

Supón que un individuo desea invertir su capital en un banco y desea saber cuánto dinero ganará después de un mes si el banco paga a razón de 2% mensual.

```
#include <stdio.h>
void main()
{
	double cap_inv=0.0,gan=0.0;
	printf("Introduce el capital invertido\n");
	scanf("%lf",&cap_inv);
	gan=(cap_inv * 0.2);
	printf("La ganancia es: %lf\n",gan);
}
```

Como podrá notarse no hay posibilidad de omitir alguna sentencia del anterior programa, todas las sentencias serán ejecutadas.

Ejercicio 3.2.

Un vendedor recibe un sueldo base más un 10% extra por comisión de sus ventas, el vendedor desea saber cuanto dinero obtendrá por concepto de comisiones por las tres ventas que realiza en el mes y el total que recibirá tomando en cuenta su sueldo base y comisiones.

```
#include <stdio.h>
void main()
{
      double tot_vta,com,tpag,sb,v1,v2,v3;
      printf("Introduce el sueldo base:\n");
      scanf("%lf",&sb);
      printf("Introduce la venta 1:\n");
      scanf("%lf",&v1);
      printf("Introduce la venta 2:\n");
      scanf("%lf",&v2);
      printf("Introduce la venta 3:\n");
      scanf("%lf",&v3);
      tot_vta = (v1+v2+v3);
      com = (tot_vta * 0.10);
      tpag = (sb + com);
      printf("total a pagar: %f\n",tpag);
}
```

Ejercicio 3.3.

Una tienda ofrece un descuento de 15% sobre el total de la compra y un cliente desea saber cuánto deberá pagar finalmente por el total de las compras.

```
#include <stdio.h>
void main()
{
	double tc,d,tp;
	printf("Introduce el total de la compra:\n");
	scanf("%f",&tc);
	d = (tc*0.15);
	tp = (tc-d);
	printf("total de las compras: %f\n",tp);
}
```

Ejercicio 3.4.

Un alumno desea saber cuál será su calificación final en la materia de Algoritmos. Dicha calificación se compone de los siguientes porcentajes:

55%, del promedio de sus tres calificaciones parciales.

30%, de la calificación del examen final.

15%, de la calificación de un trabajo final.

```
#include <stdio.h>
void main()
{
  float c1,c2,c3,ef,pef,tf,ptf,prom,ppar,cf;
  printf("Introduce la calificación 1:\n");
  scanf("%f",&c1);
  printf("Introduce la calificación 2:\n");
  scanf("%f",&c2);
  printf("Introduce la calificación 3:\n");
  scanf("%f",&c3);
  printf("Introduce la calificación del examen final:\n");
  scanf("%f",&ef);
  printf("Introduce la calificación del trabajo final:\n");
  scanf("%f",&tf);
  prom = ((c1+c2+c3)/3);
  ppar = (prom*0.55);
  pef = (ef*0.30);
  ptf = (tf*0.15);
  cf = (ppar+pef+ptf);
  printf("La calificación final es: %.2f\n",cf);
}
```

3.2. Estructura alternativa o condicional

Es aquélla en que la existencia o cumplimiento de la condición implica la **ruptura** de la **secuencia** y la ejecución de una determinada acción. Es la manera que tiene un lenguaje de programación de provocar que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos.

```
Inicio
      Si condición Entonces
          acción
     De_lo_contrario
          acción
     Fin_Si
Fin
```

IF-ELSE: La ejecución atraviesa un conjunto de estados booleanos que determinan que se ejecuten distintos fragmentos de código.

```
if (expresion-booleana)
      Sentencia1;
else
      Sentencia2;
```

```
if (expresion-booleana)
{
      Sentencia1;
      sentencia2;
}
else
      Sentencia3;
```

La cláusula **else** es opcional, la expresión puede ser de cualquier tipo y más de una (siempre que se unan mediante operadores lógicos). Otra opción posible es la utilización de **if** anidados, es decir unos dentro de otros compartiendo la cláusula **else**.

Ejercicio 3.5.

Realiza un programa que determine si un alumno aprueba o reprueba una materia.

```
# include <stdio.h>

main()
{
   float examen, tareas, trabajo, final;
   printf("Por favor introduzca la calificación de los examenes: ");
   scanf("%f",&examen);
   printf("Por favor introduzca la calificación de las tareas: ");
   scanf("%f",&tareas);
   printf("Por favor introduzca la calificación del trabajo: ");
   scanf("%f",&trabajo);
   final = (examen+tareas+trabajo)/3;
   printf("Tu calificación final es de: %.2f\n",final);
   if(final < 6)
      printf("Tendras que cursar programación nuevamente\n");
   else
      printf("Aprobaste con la siguiente calificación: %.2f\n",final);
   return(0);
}
```

SWITCH: Realiza distintas operaciones con base en el valor de la única variable o expresión. Es una sentencia muy similar a **if-else**, pero ésta es mucho más cómoda y fácil de comprender. Si los valores con los que se compara son

números se pone directamente, pero si es un carácter se debe encerrar entre comillas simples.

```
switch (expresión){
      case valor1:
            sentencia;
            break;
      case valor2:
            sentencia;
            break;
      case valor3:
            sentencia;
            break;
      case valorN:
            sentencia;
            break;
      default:
}
```

El valor de la expresión se compara con cada uno de los literales de la sentencia, **case** si coincide alguno, se ejecuta el código que le sigue, si ninguno coincide se realiza la sentencia **default** (opcional), si no hay sentencia **default** no se ejecuta nada.

La sentencia **break** realiza la salida de un bloque de código. En el caso de sentencia **switch**, realiza el código y cuando ejecuta **break**, sale de este bloque y sigue con la ejecución del programa. En el caso que varias sentencias **case** realicen la misma ejecución se pueden agrupar, utilizando una sola sentencia **break**.

```
switch (expresión){
```

```
        case valor1:
        case valor2:
        case valor5:
              sentencia;
              break;
        case valor3:
        case valor4:
              sentencia;
              break;
        default:
  }
```

Veamos un ejercicio donde se utiliza la estructura case. El usuario debe escoger cuatro opciones introduciendo un número, en caso de que el usuario introduzca un número incorrecto aparecerá el mensaje “Opción incorrecta”.

Ejercicio 3.6.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
   int opcion;
   printf("1.ALTAS\n");
   printf("2.BAJAS\n");
   printf("3.MODIFICA\n");
   printf("4.SALIR\n");
   printf("Elegir opción: ");
   scanf("%d",&opcion);

   switch(opcion)
```

```
   {
      case 1:
      printf("Opción Uno");
      break;
      case 2:
      printf("Opción Dos");
      break;
      case 3:
      printf("Opción Tres");
      break;
      case 4:
      printf("Opción Salir");
      break;
      default:
      printf("Opción incorrecta");
   }
getch();
}
```

Ejercicio 3.7.

Una persona desea saber cuánto dinero se genera por concepto de intereses sobre la cantidad que tiene en inversión en el banco. Él decidirá reinvertir los intereses siempre y cuando sean iguales o mayores a $7,000.00. Finalmente desea saber cuánto dinero tendrá en su cuenta.

```
#include <stdio.h>
void main()
{
  float p_int,cap,inte,capf;
  printf("Introduce el porcentaje de intereses:\n");
  scanf("%f",&p_int);
  printf("Introduce el capital:\n");
  scanf("%f",&cap);
   inte = (cap*p_int)/100;
  if (inte >= 7000)
   {
     capf=(cap+inte);
     printf("El capital final es: %.2f\n",capf);
   }
   else
     printf("El interes: %.2f es menor a 7000\n",inte);
}
```

Ejercicio 3.8.

Determina si un alumno aprueba a reprueba un curso, sabiendo que aprobará si su promedio de tres exámenes parciales es mayor o igual a 70 puntos; y entregó un trabajo final. (No se toma en cuenta la calificación del trabajo final, solo se toma en cuenta si lo entrego).

```
# include <stdio.h>
main()
{
   float examen1, examen2, examen3,final;
   int tra_fin=0;

   printf("Por favor introduzca la calificacion del primer
examen: ");
   scanf("%f",&examen1);
   printf("Por favor introduzca la calificacion del segundo
examen: ");
   scanf("%f",&examen2);
   printf("Por favor introduzca la calificacion del tercer
examen: ");
   scanf("%f",&examen3);
   printf("Introduzca 1 si entrego 0 si no entrego trabajo
final: ");
   scanf("%d",&tra_fin);

   final = (examen1+examen2+examen3)/3;
   if(final < 7 || tra_fin == 0)
      printf("Tendras que cursar programacion nuevamente\n");
   else
      printf("Aprobaste con la siguiente calificacion:
%.2f\n",final);
```

```
   return(0);
}
```

Ejercicio 3.9.

En un almacén se hace 20% de descuento a los clientes cuya compra supere los $1,000.00 ¿Cuál será la cantidad que pagará una persona por su compra?

```
# include <stdio.h>
main()
{
   float compra, desc, totcom;

   printf("Por Introduzca el valor de su compra: ");
   scanf("%f",&compra);

   if(compra >= 1000)
      desc = (compra*0.20);
   else
      desc = 0;

   totcom = (compra-desc);
   printf("El total de su compra es:%.2f\n",totcom);
   return(0);
}
```

Ejercicio 3.10.

Realiza una calculadora que sea capaz de sumar, restar, multiplicar y dividir, el usuario debe escoger la operación e introducir los operandos.

```
#include <stdio.h>

void main(void)
{
   int opcion;
   float op1,op2;
   printf("1.SUMAR\n");
   printf("2.RESTAR\n");
   printf("3.DIVIDIR\n");
   printf("4.MULTIPLICAR\n");
   printf("5.SALIR\n");
   printf("Elegir opción: ");
   scanf("%d",&opcion);

   switch(opcion)
   {
      case 1:
      printf("Introduzca los operandos: ");
      scanf("%f %f",&op1,&op2);
    printf("Resultado: %.2f\n",(op1+op2));
      break;
      case 2:
      printf("Introduzca los operandos: ");
      scanf("%f %f",&op1,&op2);
    printf("Resultado: %.2f\n",(op1-op2));
      break;
      case 3:
```

```
        printf("Introduzca los operandos: ");
        scanf("%f %f",&op1,&op2);
      printf("Resultado: %.2f\n",(op1/op2));
        break;
        case 4:
        printf("Introduzca los operandos: ");
        scanf("%f %f",&op1,&op2);
      printf("Resultado: %.2f\n",(op1*op2));
        break;
        case 5:
        printf("Opción Salir\n");
        break;
        default:
        printf("Opción incorrecta");
   }
}
```

3.3. Estructura repetitiva

Las estructuras repetitivas o iterativas son aquéllas en las que las acciones se ejecutan un número determinado o indeterminado de veces y dependen de un valor predefinido o el cumplimiento de una condición.

WHILE: Ejecuta repetidamente el mismo bloque de código hasta que se cumpla una condición de terminación. Hay dos partes en un ciclo: *cuerpo* y *condición*.

Cuerpo: Es la sentencia o sentencias que se ejecutarán dentro del ciclo.
Condición: Es la condición de terminación del ciclo.

```
while(condición){
      cuerpo;
}
```

Ejercicio 3.11.

Este programa convierte una cantidad en yardas, mientras el valor introducido sea mayor que 0.

```
# include <stdio.h>

main()
{
   int yarda, pie, pulgada;

   printf("Por favor deme la longitud a convertir en yardas:
");
   scanf("%d",&yarda);
   while (yarda > 0)
   {
      pulgada = 36*yarda;
      pie   = 3*yarda;
      printf("%d yarda (s) = \n", yarda);
      printf("%d pie (s) \n", pie);
      printf("%d pulgada (s) \n", pulgada);
      printf("Por favor introduzca otra cantidad a \n");
      printf("convertir (0) para terminar el programa): ");
      scanf("%d",&yarda);
   }
   printf(">>> Fin del programa <<<");
   return(0);
}
```

DO-WHILE: Es lo mismo que en el caso anterior pero aquí como mínimo siempre se ejecutará el cuerpo al menos una vez, en el caso anterior es posible que no se ejecute ni una sola vez.

```
 do{
     cuerpo;
}while(terminación);
```

Ejercicio 3.12.

Este es un ejemplo de un programa que utiliza un do-while para seleccionar una opción de un menú.

```
#include <stdio.h>
void menu(void);
main()
{
   menu();
   return(0);
}
void menu(void)
{
  char ch;

  printf("1. Opcion 1\n");
  printf("2. Opcion 2\n");
  printf("3. Opcion 3\n");
  printf("      Introduzca su opción: ");

  do {
    ch = getchar(); /* lee la selección desde el teclado */
    switch(ch) {
```

```
      case '1':
        printf("1. Opcion 1\n");
        break;
      case '2':
              printf("2. Opcion 2\n");
        break;
      case '3':
            printf("3. Opcion 3\n");
        break;
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```

Ejercicio 3.13.

Realiza un programa que lea un variable e imprima su contenido, mientras el contenido de la variable sea distinto de cero.

```
# include <stdio.h>

main()
{
   int var;

   printf("Por introduzca un valor diferente de cero: ");
   scanf("%d",&var);
   while (var != 0)
   {
      printf("Contenido de la variable: %d\n", var);
      printf("Por favor introduzca otro valor\n");
      printf("(0) para terminar el programa): ");
      scanf("%d",&var);
   }
   printf(">>> Fin del programa <<<");
   return(0);
}
```

Ejercicio 3.14.

Lee números negativos y conviértelos a positivos e imprime dichos números, mientras el número sea negativo.

```
# include <stdio.h>

main()
{
   int neg,res;

   printf("Por introduzca un valor negativo: ");
   scanf("%d",&neg);
   while (neg < 0)
   {
      res=neg*-1;
      printf("Valor positivo: %d\n",res);
      printf("Por favor introduzca otro valor\n");
      printf("Mayor o igual a (0) para terminar el programa):
");
      scanf("%d",&neg);
   }
   printf(">>> Fin del programa <<<");
   return(0);
}
```

Ejercicio 3.15.

Realiza un programa que funcione como calculadora, que utilice un menú y un while.

```
#include <stdio.h>

#define SALIDA 0
#define BLANCO ' '

void main(void)
{
   float op1,op2;
   char operador = BLANCO;

   while (operador != SALIDA)
   {
      printf("Introduzca una expresión (a (operador) b): ");
      scanf("%f%c%f", &op1, &operador, &op2);

      switch(operador)
        {
         case '+':
                  printf("Resultado: %8.2f\n",(op1+op2));
            break;
         case '-':
                  printf("Resultado: %8.2f\n",(op1-op2));
            break;
         case '*':
                  printf("Resultado: %8.2f\n",(op1*op2));
            break;
         case '/':
```

```
                printf("Resultado: %8.2f\n",(op1/op2));
          break;
       case 'x':
                operador = SALIDA;
          break;
         default: printf("Opción incorrecta");
       }
    }
}
```

FOR: Realiza las mismas operaciones que en los casos anteriores pero la sintaxis es una forma compacta. Normalmente la condición para terminar es de tipo numérico. La iteración puede ser cualquier expresión matemática válida. Si de los 3 términos que necesita no se pone ninguno se convierte en un bucle infinito.

```
for (inicio;fin;iteración)
 sentencia1;
```

```
for (inicio;fin;iteración)
{
     sentencia1;
     sentencia2;
}
```

Ejercicio 3.16.

Este programa muestra números del 1 al 100. Utilizando un bucle de tipo for.

```
#include<stdio.h>
#include<conio.h>
void main(void)
{
      int n1=0;
      for (n1=1;n1<=20;n1++)
         printf("%d\n",n1);
      getch();
```

```
}
```

Ejercicio 3.17.

Calcula el promedio de un alumno que tiene 7 calificaciones en la materia de Introducción a la programación.

```
#include<stdio.h>
void main(void)
{
int n1=0;
      float cal=0,tot=0;
      for (n1=1;n1<=7;n1++)
      {
            printf("Introduzaca la calificación %d:",n1);
            scanf(" %f",&cal);
           tot=tot+cal;
      }
      printf("La calificación es: %.2f\n",tot/7);
}
```

Ejercicio 3.18.

Lee 10 números y obtén su cubo y su cuarta.

```
#include<stdio.h>
void main(void)
{
   int n1=0,num=0,cubo,cuarta;
   for (n1=1;n1<=10;n1++)
   {
      printf("Introduzca el numero %d:",n1);
      scanf(" %d",&num);
      cubo=num*num*num;
      cuarta=cubo*num;
      printf("El cubo de %d es: %d La cuarta es:
%d\n",num,cubo,cuarta);
   }
}
```

Ejercicio 3.19.

Lee 10 números e imprime solamente los números positivos.

```
#include<stdio.h>
void main(void)
{
      int n1=0,num=0;
      for (n1=1;n1<=10;n1++)
      {
            printf("Introduzca el numero %d:",n1);
            scanf(" %d",&num);
            if (num > 0)
                  printf("El numero %d es positivo\n",num);
      }
}
```

CONCLUSIONES

Las estructura secuencial permite la realización de programas que no requieren tomar decisión alguna, y es la manera más sencilla de realizar un programa.

Las estructura alternativa permite que un programa se comporte de acuerdo a los datos que han sido introducidos por el usuario, o datos que han sido generados dentro del programa.

El ciclo de tipo **FOR** permite ejecutar un número de sentencias un número determinado de veces.

Por último Los ciclos while son útiles cuando se desconoce el número de iteraciones que serán ejecutadas en el programa.

.

Bibliografía del tema 3

www.lawebdelprogramador.com

Actividades de aprendizaje

A.3.1. Realiza un programa que permita la captura de 10 números y determine cuántos son negativos, cuántos positivos y cuántos son ceros.

A.3.2. Realiza un programa que pida una edad. Si la edad es igual o menor a 10 que muestre el mensaje NIÑO, si la edad es mayor a 65 que muestre el mensaje JUBILADO, y si la edad es mayor a 10 y menor o igual 65 que muestre el mensaje ADULTO.

A.3.3. Una persona invierte $5,000.00 en un banco a razón de 1% mensual. Calcula el monto al final del año, recuerda que debe acumular el interés de cada mes al monto.

A.3.4. Realizar un programa que determine la edad de una persona con base en la fecha de nacimiento que introduzca el usuario, mientras la fecha de nacimiento sea mayor a 1950, utiliza un ciclo WHILE.

A.3.5. Realizar un programa que muestre la tabla de multiplicar de un número que introduzca el usuario, utiliza un ciclo FOR.

Cuestionario de autoevaluación

1. Explica la estructura secuencial.
2. Explica la estructura alterativa.
3. Explica la estructura repetitiva.
4. ¿Cuál es la diferencia entre un ciclo *while* y un ciclo *for*?
5. ¿Cuál es la diferencia entre un ciclo *while* y un ciclo *do.while*
6. ¿Cuál la función de la palabra *if*?
7. ¿Cuál es la función de la palabra *else*?
8. ¿Cuál es la función de la palabra *switch*?
9. ¿Cuál es la función de la palabra *case*?
10. ¿Cuál es la función de la palabra *break*?

Examen de autoevaluación

1. La estructura secuencial permite:
   a. Que todas las sentencias se ejecuten
   b. Que se ejecuten solo algunas sentencias
   c. Que se ejecuten de acuerdo a una condición
   d. Que se ejecuten un número determinado de veces

2. La estructura alternativa permite que:
   a. Un flujo se bifurque
   b. El uso de iteraciones
   c. Un flujo no se bifurque
   d. El uso de sentencias repetitivas

3. La siguiente es una palabra reservada que se usa en la estructura alternativa:
   a. *do*
   b. *if*
   c. *for*
   d. *main()*

4. La siguiente es una palabra reservada que se usa en la estructura switch:
    a. *default*
    b. *while*
    c. *for*
    d. *else*

5. Si se desea ejecutar más de una sentencia en un ciclo, estas sentencias deben:
    a. Estar entre llaves
    b. Estar sin llaves
    c. Estar entre paréntesis
    d. Estar sin paréntesis

6. Un ciclo de tipo for es:
    a. Un ciclo con un número determinado de iteraciones
    b. Un ciclo con un número indeterminado de iteraciones
    c. Un ciclo infinito
    d. Un ciclo de tipo do-while

7. i++ significa que:
    a. i es una constante
    b. i se decrementa en uno
    c. i se incrementa en uno
    d. i es una palabra reservada

8. La siguiente es una palabra reservada que se usa en la estructura repetitiva:
    a. *break*
    b. *default*
    c. *while*
    d. *else*

9. Un ciclo while:
    a. Es una función.
    b. Es una constante.
    c. Tiene un número indeterminado de iteraciones.
    d. Se ejecuta al menos una vez.

10. Un ciclo do-while:
    a. Se ejecuta solo si se cumple la condición.
    b. Se ejecuta al menos una vez.
    c. Es un ciclo con un número indeterminado de iteraciones.
    d. Es un tipo de case.

TEMA 4. FUNCIONES

Objetivo particular

Se conocerá e identificará la forma en que trabajan las funciones en el lenguaje C.

Temario detallado

4.1. Internas
- 4.1.1. Funciones de caracteres y cadenas
- 4.1.2. Funciones matemáticas
- 4.1.3. Funciones de conversión

4.2. Definidas por el usuario

4.3. Ámbito de variables (locales y globales)

4.4. Recursividad

Introducción

Las funciones son los bloques constructores de C, es el lugar donde se produce toda la actividad del programa. Las funciones realizan una tarea específica, como mandar a imprimir un mensaje en pantalla, o abrir un archivo.

Es la característica más importante de C. Subdivide en varias tareas el programa, así sólo se las tendrá que ver con diminutas piezas de programa, de pocas líneas, cuya escritura y corrección es una tarea simple. Las funciones pueden o no devolver y recibir valores del programa.

4.1. Internas

El lenguaje C cuenta con funciones internas que realizan tareas específicas. Por ejemplo, hay funciones para el manejo de caracteres y cadenas.

4.1.1. Funciones de caracteres y cadenas

La biblioteca estándar de C tiene un variado conjunto de funciones de manejo de caracteres y cadenas. En una implementación estándar, las funciones de cadena

requieren el archivo de cabecera **string.h**, que proporciona sus prototipos. Las funciones de caracteres utilizan **ctype.h** como archivo de cabecera.

Veamos las funciones para el manejo de caracteres.

isalpha: Devuelve un entero. DISTINTO DE CERO si la variable es una letra del alfabeto, en caso contrario devuelve cero. Cabecera: **<ctype.h>**.

```
int isalpha(variable_char);
```

isdigit: Devuelve un entero. DISTINTO DE CERO si la variable es un número (0 a 9), en caso contrario devuelve cero. Cabecera **<ctype.h>**.

```
int isdigit(variable_char);
```

isgraph: Devuelve un entero. DISTINTO DE CERO si la variable es cualquier carácter imprimible distinto de un espacio, si es un espacio CERO. Cabecera **<ctype.h>**.

```
int isgraph(variable_char);
```

islower: Devuelve un entero. DISTINTO DE CERO si la variable está en minúscula, en caso contrario devuelve cero. Cabecera **<ctype.h>**.

```
int islower(variable_char);
```

ispunct: Devuelve un entero. DISTINTO DE CERO si la variable es un carácter de puntuación, en caso contrario, devuelve cero. Cabecera **<ctype.h>**

```
int ispunct(variable_char);
```

isupper: Devuelve un entero. DISTINTO DE CERO si la variable está en mayúsculas, en caso contrario, devuelve cero. Cabecera **<ctype.h>**

```
int isupper(varible_char);
```

Veamos un ejercicio donde se utiliza la función isalpha.

Ejercicio 4.1.

Cuenta el número de letras y números que hay en una cadena. La longitud debe ser siempre de cinco, por no conocer aún la función que me devuelve la longitud de una cadena.

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
void main(void)
{
      int ind,cont_num=0,cont_text=0;
      char temp;
      char cadena[6];
      printf("Introducir 5 caracteres: ");
      gets(cadena);
      for(ind=0;ind<5;ind++)
      {
            temp=isalpha(cadena[ind]);
            if(temp)
                              cont_text++;
            else
                  cont_num++;
      }
      printf("El total de letras es  %d\n",cont_text);
```

```
        printf("El total de numeros es %d",cont_num);
        getch();
}
```

Si la función isalpha devuelve in entero distinto de cero, se incrementa la variable cont_text, de lo contrario se incrementa la variable cont_num.

El ejercicio anterior muestra el uso de funciones para el manejo de caracteres, veamos las funciones para el manejo de cadenas.

memset: Inicializar una región de memoria (buffer) con un valor determinado. Se utiliza principalmente para inicializar cadenas con un valor determinado. Cabecera **<string.h>**.

```
memset (var_cadena,'carácter',tamaño);
```

strcat: Concatena cadenas, es decir, añade la segunda a la primera, que no pierde su valor original. Hay que tener en cuenta que la longitud de la primera cadena debe ser suficiente para guardar la suma de las dos cadenas. Cabecera **<string.h>**.

```
strcat(cadena1,cadena2);
```

strchr: Devuelve un puntero a la primera ocurrencia del carácter especificado en la cadena donde se busca. Si no lo encuentra, devuelve un puntero nulo. Cabecera **<string.h>**.

```
strchr(cadena,'carácter');
strchr("texto",'carácter');
```

strcmp: Compara alfabéticamente dos cadenas y devuelve un entero basado en el resultado de la comparación. La comparación no se basa en la longitud de las cadenas. Muy utilizado para comprobar contraseñas. Cabecera **<string.h>**.

```
strcmp(cadena1,cadena2);
strcmp(cadena2,"texto");
```

RESULTADO	
VALOR	DESCRIPCIÓN
Menor a Cero	Cadena1 menor a Cadena2.
Cero	Las cadenas son iguales.
Mayor a Cero	Cadena1 mayor a Cadena2.

strcpy: Copia el contenido de la segunda cadena en la primera. El contenido de la primera se pierde. Lo único que debemos contemplar es que el tamaño de la segunda cadena sea menor o igual al tamaño de la primer cadena. Cabecera **<string.h>**.

```
strcpy(cadena1,cadena2);
strcpy(cadena1,"texto");
```

strlen: Devuelve la longitud de la cadena terminada en nulo. El carácter nulo no se contabiliza. Devuelve un valor entero que indica la longitud de la cadena. Cabecera **<string.h>**.

```
variable=strlen(cadena);
```

strncat: Concatena el número de caracteres de la segunda cadena en la primera. Ésta última no pierde la información. Hay que controlar que la

longitud de la primera cadena tenga longitud suficiente para guardar las dos cadenas. Cabecera **<string.h>**.

```
strncat(cadena1,cadena2,nº de caracteres);
```

strncmp: Compara alfabéticamente un número de caracteres entre dos cadenas. Devuelve un entero según el resultado de la comparación. Los valores devueltos son los mismos que en la función strcmp. Cabecera **<string.h>**.

```
strncmp(cadena1,cadena2,nº de caracteres);
strncmp(cadena1,"texto",nº de caracteres);
```

strncpy: Copia un número de caracteres de la segunda cadena a la primera. En la primera cadena se pierden aquellos caracteres que se copian de la segunda. Cabecera **<string.h>**.

```
strncpy(cadena1,cadena2,nº de caracteres);
strncpy(cadena1,"texto",nº de caracteres);
```

strrchr: Devuelve un puntero a la última ocurrencia del carácter buscado en la cadena. Si no lo encuentra devuelve un puntero nulo. Cabecera **<string.h>**.

```
strrchr(cadena,'carácter');
strrchr("texto",'carácter');
```

strpbrk: Devuelve un puntero al primer carácter de la cadena que coincida con otro de la cadena a buscar. Si no hay correspondencia devuelve un puntero nulo. Cabecera **<string.h>**.

```
strpbrk("texto","cadena_de_busqueda");

strpbrk(cadena,cadena_de_busqueda);
```

tolower: Devuelve el carácter equivalente al de la variable en minúsculas si la variable es una letra; y no la cambia si la letra es minúscula. Cabecera **<ctype.h>**.

```
variable_char=tolower(variable_char);
```

toupper: Devuelve el carácter equivalente al de la variable en mayúsculas si la variable es una letra; y no la cambia si la letra es minúscula. Cabecera **<ctype.h>**.

```
variable_char=toupper(variable_char);
```

Ejercicio 4.2.

Este programa busca la primera coincidencia y muestra la cadena a partir de esa coincidencia.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

void main(void)
{
      char letra;
      char *resp;
      char cad[30];

      printf("Cadena: ");
      gets(cad);
      printf("Buscar Letra: ");
      letra=getchar();

      resp=strchr(cad,letra);
```

```
        if(resp)
              printf("%s",resp);
        else
              printf("No Esta");

        getch();
}
```

Ejercicio 4.3.

Este programa busca la última coincidencia y muestra la cadena a partir de ese punto.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

void main(void)
{
        char letra;
        char *resp;
        char cad[30];

        printf("Cadena: ");
        gets(cad);
        printf("Buscar Letra: ");
        letra=getchar();

        resp=strrchr(cad,letra);

        if(resp)
              printf("%s",resp);
        else
              printf("No Esta");

        getch();
}
```

Ejercicio 4.4.

En este ejemplo se busca en una cadena a partir de un grupo de caracteres que introduce el usuario. Si encuentra alguna coincidencia, la imprime en pantalla, de lo contrario muestra un mensaje de error en pantalla.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

void main(void)
{
      char letras[5];
      char *resp;
      char cad[30];
      printf("Introducir cadena: ");
      gets(cad);
      printf("Posibles letras(4): ");
      gets(letras);
      resp=strpbrk(cad,letras);
      if(resp)
            printf("%s",resp);
      else
            printf("Error");
      getch();
}
```

4.1.2. Funciones matemáticas

El estándar C define 22 funciones matemáticas que entran en las siguientes categorías**, trigonométricas, hiperbólicas, logarítmicas**, **exponenciales** y otras. Todas las funciones requieren el archivo de cabecera **math.h**.

acos: Devuelve un tipo *double*. Muestra el arcocoseno de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera **<math.h>**.

```
double acos(variable_double);
```

asin: Devuelve un tipo *double*. Muestra el arcoseno de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera **<math.h>**.

```
double asin(variable_double);
```

atan: Devuelve un tipo *double*. Muestra el arcotangente de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera **<math.h>**.

```
double atan(variable_double);
```

cos: Devuelve un tipo *double*. Muestra el coseno de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera **<math.h>**.

```
double cos(variable_double_radianes);
```

sin: Devuelve un tipo *double*. Muestra el seno de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera **<math.h>**.

```
double sin(variable_double_radianes);
```

tan: Devuelve un tipo *double*. Muestra la tangente de la variable, ésta debe ser de tipo *double* y estar expresada en radianes. Cabecera **<math.h>**.

```
double tan(variable_double_radianes);
```

cosh: Devuelve un tipo *double*. Muestra el coseno hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser **<math.h>**.

```
double cosh(variable_double);
```

sinh: Devuelve un tipo *double*. Muestra el seno hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser **<math.h>**.

```
double sinh(variable_double);
```

tanh: Devuelve un tipo *double*. Muestra la tangente hiperbólico de la variable, ésta debe ser de tipo *double* y estar en el rango –1 y 1, en otro caso se produce un error de dominio. Cabecera debe ser **<math.h>**.

```
double tanh(variable_double);
```

ceil: Devuelve un *double* que representa el menor entero sin serlo más que la variable redondeada. Por ejemplo, dado 1.02 devuelve 2.0. Si asignamos –1.02, devuelve –1. En resumen, redondea la variable a la alta. Cabecera **<math.h>**.

```
double ceil(variable_double);
```

floor: Devuelve un *double* que representa el entero mayor, sin serlo más quela variable redondeada. Por ejemplo dado 1.02 devuelve 1.0. Si asignamos –1.2 devuelve –2. En resumen redondea la variable a la baja. Cabecera **<math.h>**.

```
double floor(variable_double);
```

fabs: Devuelve un valor *float* o *double*. Devuelve el valor absoluto de una variable *float*. Se considera una función matemática, pero su cabecera es **<stdlib.h>**.

```
var_float fabs(variable_float);
```

labs: Devuelve un valor *long*. Devuelve el valor absoluto de una variable long. Se considera una función matemática, pero su cabecera es **<stdlib.h>**.

```
var_long labs(variable_long);
```

abs: Devuelve un valor entero. Devuelve el valor absoluto de una variable int. Se considera una función matemática, pero su cabecera es **<stdlib.h>**.

```
var_float abs(variable_float);
```

modf: Devuelve un *double*. Descompone la variable en su parte entera y fraccionaria. La parte decimal es el valor que devuelve la función, su parte entera la guarda en el segundo término de la función. Las variables tienen que ser obligatoriamente de tipo *double*. Cabecera **<math.h>**.

```
var_double_decimal= modf(variable,var_parte_entera);
```

pow: Devuelve un *double*. Realiza la potencia de un número base elevado a un exponente que nosotros le indicamos. Se produce un error si la base es cero o el exponente es menor o igual a cero. La cabecera es **<math.h>**.

```
var_double=pow(base_double,exponente_double);
```

sqrt: Devuelve un *double*. Realiza la raíz cuadrada de la variable, ésta no puede ser negativa, si lo es se produce un error de dominio. La cabecera **<math.h>**.

```
var_double=sqrt(variable_double);
```

log: Devuelve un *double*. Realiza el logaritmo natural (neperiano) de la variable. Se produce un error de dominio si la variable es negativa y un error de rango si es cero. Cabecera **<math.h>**.

```
double log(variable_double);
```

log10: Devuelve un valor *double*. Realiza el logaritmo decimal de la variable de tipo double. Se produce un error de dominio si la variable es negativa y un error de rango si el valor es cero. La cabecera que utiliza es **<math.h>**.

```
double log10(var_double);
```

ramdomize(): Inicializa la semilla para generar números aleatorios. Utiliza las funciones de tiempo para crear esa semilla. Esta función está relacionada con random. Cabecera **<stdlib.h>**.

```
void randomize();
```

random: Devuelve un entero. Genera un número entre 0 y la variable menos uno. Utiliza el reloj del ordenador para ir generando esos valores. El programa debe llevar la función randomize para cambiar la semilla en cada ejecución. Cabecera **<stdlib.h>**.

```
int random(variable_int);
```

Ejercicio 4.5.

Este programa imprime las diez primeras potencias de 10

```
#include <math.h>
#include <stdio.h>

int main(void)
{
  double x = 10.0, y = 0.0;

  do {
    printf("%f\n", pow(x, y));
    y++;
  } while(y<11.0);

  return 0;
}
```

Como puede observarse en el programa, se utiliza la función **pow** para obtener la potencia de un número.

4.1.3. Funciones de conversión

En el estándar de C se definen funciones para realizar conversiones entre valores numéricos y cadenas de caracteres. La cabecera de todas estas funciones es **stdlib.h**. Se pueden dividir en dos grupos, conversión de valores numéricos a cadena y conversión de cadena a valores numéricos.

atoi: Devuelve un entero. Esta función convierte la cadena en un valor entero. La cadena debe contener un número entero válido, si no es así el valor devuelto queda indefinido. La cadena puede terminar con espacios en

blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es **<stdlib.h>**.

```
int atoi(variable_char);
```

atol: Devuelve un *long*. Esta función convierte la cadena en un valor *long*. La cadena debe contener un número long válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es **<stdlib.h>**.

```
long atol(variable_char);
```

atof: Devuelve un *double*. Esta función convierte la cadena en un valor *double*. La cadena debe contener un número *double* válido, si no es así, el valor devuelto queda indefinido. La cadena puede terminar con espacios en blanco, signos de puntuación y otros que no sean dígitos, la función los ignora. La cabecera es **<stdlib.h>**.

```
double atof(variable_char);
```

sprintf: Devuelve una cadena. Esta función convierte cualquier tipo numérico a cadena. Para convertir de número a cadena hay que indicar el tipo de variable numérica y tener presente que la longitud de la cadena debe poder guardar la totalidad del número. Admite también los formatos de salida, es decir, que se pueden coger distintas partes del número. La cabecera es **<stdlib.h>**.

```
sprintf(var_cadena,"identificador",var_numerica);
```

itoa: Devuelve una cadena. La función convierte un entero en su cadena equivalente y sitúa el resultado en la cadena definida en segundo término de la función. Hay que asegurarse de que la cadena sea lo suficientemente grande para guardar el número. Cabecera **<stdlib.h>**.

```
itoa(var_entero,var_cadena,base);
```

BASE	DESCRIPCIÓN
2	Convierte el valor en binario.
8	Convierte el valor a Octal.
10	Convierte el valor a decimal.
16	Convierte el valor a hexadecimal.

ltoa: Devuelve una cadena. La función convierte un long en su cadena equivalente y sitúa el resultado en la cadena definida en segundo término de la función. Hay que asegurarse de que la cadena sea lo suficientemente grande para guardar el número. Cabecera **<stdlib.h>**.

```
ltoa(var_long,var_cadena,base);
```

Existen otras bibliotecas en C, además de las vistas, por ejemplo:

- entrada y salida de datos (stdio.h)
- memoria dinámica (stdlib.h)

4.2. Definidas por el usuario

El mecanismo para trabajar con funciones es el siguiente, primero debemos declarar el **prototipo** de la función, a continuación debemos hacer la **llamada** y por último desarrollar la función. Los dos últimos pasos pueden cambiar, es decir, no es necesario que el desarrollo de la función esté debajo de la llamada.

Antes de seguir debemos conocer las reglas de ámbito de las funciones. El código de una función es privado a ella, el código que comprende su cuerpo está oculto al resto del programa a menos que se haga a través de una llamada. Todas las variables que se definen dentro de una función son locales con excepción de las variables estáticas.

SINTAXIS DEL PROTOTIPO:

```
tipo_devuelto nombre_funcion ([parametros]);
```

SINTAXIS DE LA LLAMADA:

```
nombre_funcion([parametros]);
```

SINTAXIS DEL DESARROLLO:

```
tipo_devuelto nombre_funcion ([parametros])
{
      cuerpo;
}
```

Ejercicio 4.5.

Un mensaje es desplegado a través de una función.

```
#include <stdio.h>
void saludo();
void primer_mensaje();
void main ( )
{
   saludo();
   primer_mensaje();
}

void saludo()
{
   printf ("Buenos dias\n");
```

```
}
void primer_mensaje()
{
   printf("Un programa esta formado ");
   printf("por funciones\n");
}
```

La función **primer_mensaje** despliega un mensaje en pantalla, esta función es llamada desde la función principal o **main()**.

Cuando se declaran las funciones es necesario informar al compilador el tamaño de los valores que se le enviarán y el tamaño del valor que retorna. En el caso de no indicar nada para el valor devuelto toma por defecto el valor int.

Al llamar a una función se puede hacer la llamada **por valor** o **por referencia**. En el caso de hacerla por valor se copia el contenido del argumento al parámetro de la función, es decir, si se producen cambios en el parámetro no afecta a los argumentos. C utiliza esta llamada por defecto. Cuando se utiliza la llamada por referencia lo que se pasa a la función es la dirección de memoria del argumento, por tanto, si se producen cambios, afectan también al argumento. La llamada a una función se puede hacer tantas veces como se quiera.

PRIMER TIPO: Las funciones de este tipo ni devuelven valor ni se les pasan parámetros. En este caso hay que indicarle que el valor que devuelve es de tipo void, y para indicarle que no recibirá parámetros también utilizamos el tipo void. Cuando realizamos la llamada no hace falta indicarle nada, se abren y cierran los paréntesis.

```
void nombre_funcion(void);
nombre_funcion();
```

Ejercicio 4.6.

El siguiente programa es muy similar al anterior, este tipo de funciones no tienen parámetros ni devuelven ningún valor.

```
#include <stdio.h>
#include <conio.h>
void mostrar(void);
void main(void)
{
	printf("Estoy en la principal\n");
	mostrar();
	printf("De vuelta en la principal");
	getch();
}

void mostrar(void)
{
	printf("Ahora he pasado a la función, presione cualquier tecla\n");
	getch();
}
```

SEGUNDO TIPO: Son funciones que devuelven un valor una vez que han terminado de realizar sus operaciones, sólo se puede devolver uno. La devolución se realiza mediante la sentencia **return**, que además de devolver un valor hace que la ejecución del programa vuelva al código que llamó a esa función. Al compilador hay que indicarle el tipo de valor que se va a devolver poniendo delante del nombre de la función el tipo a devolver. En este tipo de casos la función es como si fuera una variable, pues toda ella equivale al valor que devuelve.

```
tipo_devuelto nombre_funcion(void);
```

```
variable=nombre_funcion();
```

Ejercicio 4.7.

Este programa calcula la suma de dos números introducidos por el usuario, la función **suma()** devuelve el resultado de la operación a la función principal a través de la función **return()**.

```
#include <stdio.h>
#include <conio.h>
int suma(void);
void main(void)
{
      int total;
      printf("Suma valores\n");
      total=suma();
      printf("\n%d",total);
      getch();
}

int suma(void)
{
      int a,b,total_dev;
      printf("valores: ");
      scanf("%d %d",&a,&b);
      total_dev=a+b;
      return total_dev;
}
```

TERCER TIPO: En este tipo las funciones pueden o no devolver valores pero lo importante es que las funciones pueden recibir valores. Hay que indicar al compilador cuántos valores recibe y de qué tipo es cada uno de ellos. Se le indica poniéndolo en los paréntesis que tienen las funciones. Deben ser los mismos que en el prototipo.

```
void nombre_funcion(tipo1,tipo2…tipoN);
nombre_funcion(var1,var2…varN);
```

Ejercicio 4.8.
Este programa utiliza una función de nombre **resta()**, que tiene dos parámetros, los parámetros son los operandos de una resta, además la función devuelve el resultado de la operación a la función principal a través de la función **return()**.

```
#include<stdio.h>

int resta(int x, int y); /*prototipo de la función*/

main()
{
	int a=5;
	int b=93;
	int c;

	c=resta(a,b);
	printf("La diferencia es: %d\n",c);
	return(0);
}

int resta(int x, int y) /*declaracion de la función*/
{
	int z;

	z=y-x;
	return(z);	/*tipo devuelto por la función*/
}
```

Ejercicio 4.9.

Realiza un programa que lea un número entero y determine si es par o impar.

```
/*Lee un numero entero y determina si es par o impar */
#include <stdio.h>
#define MOD %
/* %, es el operador que obtiene el resto de la división
entera */
#define EQ ==
#define NE !=
#define SI 1
#define NO 0
void main ( )
{
      int n, es_impar(int);
      printf ("Introduzca un entero: \n");
      scanf ("%d", &n);
      if ( es_impar (n) EQ SI )
            printf ("El numero %d es impar. \n", n);
      else
            printf ("El numero %d no es impar. \n", n);
}

int es_impar (int x)
{
      int respuesta;
      if ( x MOD 2 NE 0 ) respuesta=SI;
      else
            respuesta=NO;
      return (respuesta);
}
```

Ejercicio 4.10.

Realiza un programa en C que sume, reste, multiplicaque y divida con datos introducidos por el usuario.

```
#include<stdio.h>

int resta(int x, int y); /*prototipo de la función*/
int suma(int x, int y);  /*prototipo de la función*/

main()
{
      int a,b,c,d;
      printf("Introduce el valor de a: ");
      scanf("%d",&a);
      printf("\nIntroduce el valor de b: ");
      scanf("%d",&b);
      c=resta(a,b);
      d=suma(a,b);
      printf("La diferencia es: %d\n",c);
      printf("La suma es: %d\n",d);
      return(0);
}

int resta(int x, int y) /*declaración de la función*/
{
      int z;
      z=y-x;
      return(z);      /*tipo devuelto por la función*/
}

int suma(int x, int y) /*declaración de la función*/
{
      int z;
      z=y+x;
      return(z);      /*tipo devuelto por la función*/
}
```

Ejercicio 4.11.

Realiza una función que eleve un número al cubo.

```
#include<stdio.h>

int cubo(int base);
void main(void)
{
	int res,num;
	printf("Introduzca un numero:");
	scanf(" %d",&num);
	res=cubo(num);
	printf("El cubo de %d es: %d\n",num,res);
}

int cubo(int x)
{
	int cubo_res;
	cubo_res=x*x*x;
	return(cubo_res);
}
```

Ejercicio 4.12.

Realiza una función que eleve un número a un exponente cualquiera.

```
#include<stdio.h>

int cubo(int base,int expo);
void main(void)
{
	int res,num,ex;
	printf("Introduzca una base:");
	scanf(" %d",&num);
	printf("Introduzca un exponente:");
	scanf(" %d",&ex);
	res=cubo(num,ex);
	printf("El numero %d elevado es: %d\n",num,res);
}

int cubo(int x, int y)
{
	int i,acum=1;
	for(i=1;i<=y;i++)
		acum=acum*x;
```

```
        return(acum);
}
```

Ejercicio 4.13.

Realiza una función que obtenga la raíz cuadrada de un número.

```
# include <stdio.h>
# include <math.h>

void impresion(void);

main()
{
        printf("Este programa extraerá una raiz
cuadrada.\n\n");
        impresion();
        return(0);
}

void impresion(void)
{
        double z=4;
        double x;
        x=sqrt(z);
        printf("La raíz cuadrada de %lf es %lf \n",z,x);
}
```

Ejercicio 4.14.

El **método de la burbuja** es un algoritmo de ordenación de los más sencillos, busca el arreglo desde el segundo hasta el último elemento comparando cada uno con el que le precede. Si el elemento que le precede es mayor que el elemento actual, los intercambia de tal modo que el más grande quede más cerca del final del arreglo. En la primera pasada, esto resulta en que el elemento más grande termina al final del arreglo.

El siguiente ejercicio usa un arreglo de diez elementos desordenados y utiliza la función **bubble** para ordenar los elementos. La función **bubble** compara un elemento con el siguiente, si es mayor, entonces los intercambia, para eso usa

una variable temporal de nombre **t**. El proceso se repite un número de veces igual al número de elementos menos uno. El resultado final es un arreglo ordenado.

```
/* Programa de ordenamiento por burbuja */
#include <stdio.h>
int arr[10] = {3,6,1,2,3,8,4,1,7,2};
void bubble(int a[], int N);
int main(void)
{
     int i;
     putchar('\n');
     for (i = 0; i < 10; i++)
     {
          printf("%d ", arr[i]);
     }
     bubble(arr,10);
     putchar('\n');
     for (i = 0; i < 10; i++)
     {
          printf("%d ", arr[i]);
     }
     return 0;
}

void bubble(int a[], int N)
{
     int i, j, t;
     for (i = N-1; i >= 0; i--)
     {
          for (j = 1; j <= i; j++)
          {
               if (a[j-1] > a[j])
               {
                    t = a[j-1];
                    a[j-1] = a[j];
                    a[j] = t;
               }
          }
     }
}
```

4.3. Ámbito de variables (locales y globales)

Las variables son locales cuando se declaran dentro de una función. Las variables locales sólo pueden ser referenciadas (utilizadas) por sentencias que estén dentro de la función donde han sido declaradas.

Las variables son globales cuando son conocidas a lo largo de todo el programa, y se pueden usar desde cualquier parte del código. Mantienen sus valores durante toda la ejecución. Deben ser declaradas fuera de todas las funciones incluida **main()**.

4.4. Recursividad

La recursividad es el proceso de definir algo en términos de sí mismo, es decir, que las funciones pueden llamarse a sí mismas, esto se consigue cuando en el cuerpo de la función hay una llamada a la propia función, se dice que es recursiva. Una función recursiva no hace una nueva copia de la función, sólo son nuevos los argumentos.

La principal ventaja de las funciones recursivas es que se pueden usar para crear versiones de algoritmos más claras y sencillas. Cuando se escriben funciones recursivas, se debe tener una sentencia **if** para forzar a la función a volver sin que se ejecute la llamada recursiva.

Ejercicio 4.14.
Realizar un programa que obtenga el factorial de un número utilizando una función recursiva. Como podras observar dentro de la función **factorial**, se hace una nueva llamada a la función, esto es un llmada recursiva.

```
#include <stdio.h>
double factorial(double respuesta);
main()
{
```

```
        double numero=3.0;
        double fact;
        fact=factorial(numero);
        printf("El factorial vale: %15.0lf \n",fact);
        return(0);
}

double factorial(double respuesta)
{
        if (respuesta <= 1.0)
                return(1.0);
        else
                return(respuesta*factorial(respuesta-1.0));
}
```

La función trabaja de la siguiente manera, en cada iteración el valor de respuesta se decrementa en 1.

El valor de factorial es multiplicado por el valor de respuesta en cada iteración.
En la primera iteración el valor de respuesta es 3, después 2 y por último 1.
En la primera iteración se multiplica por (3-1), y después (2-1).

Las funciones recursivas pueden ahorrar la escritura de codigo, sin embargo se deben usar con precaución, pues pueden generar un excesivo consumo de memoria.

CONCLUSIONES

Las funciones permiten el desarrollo de tareas específicas dentro de un programa.
Las funciones para el manejo de caracteres son útiles en el caso de que se desee conocer las características de caracteres individuales.

Las funciones para el manejo de cadenas permiten su comparación o modificación.

Las funciones de para el manejo de fecha y hora permiten obtener la fecha y hora que mantiene el sistema operativo, son muy utilizadas para el desarrollo de tareas programadas.

Las funciones matemáticas permiten el calculo de diversas operaciones, esto hace que el programador pueda desarrollar mas rápido programas que requieran de cálculos matemáticos.

C permite el desarrollo de funciones propias, lo que lo hace un lenguaje muy adaptable a las necesidades del programador.

Bibliografía del tema 4

www.lawebdelprogramador.com

Actividades de aprendizaje

A.4.1. Realiza una función que determine si dos cadenas introducidas por el usuario son iguales.

A.4.2. Realiza una función que determine si un número introducido por el usuario es par o impar.

A.4.3. Realiza una función que cambie las letras minúsculas a mayúsculas.

A.4.4. Realiza una función que determine la edad de una persona, con base en su fecha de nacimiento.

A.4.5. Realiza una función que transforme una cantidad introducida en pesos, a su equivalente en dólares y euros.

Cuestionario de autoevaluación

1. ¿Qué es una función?
2. ¿Qué es una librería?
3. ¿Qué es un carácter?

4. ¿Qué es una cadena?
5. ¿Qué es una función interna?
6. ¿Qué es una función definida por el usuario?
7. ¿Qué es un parámetro?
8. ¿Cuál es la función principal en un programa en C?
9. ¿Cuántos valores puede devolver una función?
10. ¿Qué es una función recursiva?

Examen de autoevaluación

1. *void* indica que:
   a. u ciclo
   b. Que la función no devuelve un valor
   c. Una función
   d. Que la función devuelve valores

2. La palabra reservada return sirve para:
   a. Regresar parametros.
   b. Devolver un valor.
   c. Guardar un valor.
   d. Indica los parámetros de una función.

3. Si encuentra un número, la función isdigit devuelve:
   a. Un cero
   b. Un entero distinto de cero
   c. Un numero
   d. Un carácter

4. La función srtcat:
    a. Cambia una letra a mayúscula.
    b. Une dos cadenas.
    c. Inicializa un arreglo.
    d. Copia una cadena sobre otra.

5. La función strcmp:
    a. Compara dos cadenas.
    b. Busca la primera aparición de un carácter.
    c. Busca la ultima aparición de un carácter.
    d. Busca la coincidencia de un carácter en dos cadenas.

6. Las cadenas se almacenan en:
    a. Constantes.
    b. Arreglos.
    c. Estructuras.
    d. Listas enlazadas.

7. La función isgraph:
    a. Determina si un carácter es imprimible
    b. Se utiliza en la realización de gráficas
    c. Obtiene la fecha del sistema
    d. Determina si un carácter es una letra

8. La función ceil:
    a. Redondea un número
    b. Obtiene el seno de un número
    c. Obtiene el logaritmo de un número
    d. Obtiene el coseno de un número

9. La función atof:
    a. Convierte una cadena en un double
    b. Devuelve un entero
    c. Devuelve un valor
    d. Se usa en las funciones de primer tipo

10. La recursividad permite:
    a. Que una función se llame a si misma.
    b. Funciones con parámetros.
    c. Funciones que devuelven valores.
    d. Funciones que usan return().

TEMA 5. ARREGLOS Y ESTRUCTURAS

Objetivo particular

El alumno comprenderá el concepto de arreglo unidimensional y multidimensional, además del uso de arreglos y cadenas. También conocerá el uso de las estructuras y enumeraciones.

Temario detallado

5.1. Arreglos unidimensionales
5.2. Arreglos multidimensionales
5.3. Arreglos y cadenas
5.4. Estructuras

Introducción

Un **arreglo** es una colección de variables del mismo tipo que se referencian por un nombre en común. A un elemento específico de un arreglo se accede mediante un índice. Todos los arreglos constan de posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento. Los arreglos pueden tener una o varias dimensiones.

Una estructura es una colección de variables que se referencia bajo un único nombre.

Por último la enumeración es un conjunto de constantes enteras con nombre que especifica todos los valores válidos que una variable de ese tipo puede tener.

5.1. Arreglos unidimensionales

El arreglo más común en C es la cadena (arreglo de caracteres terminado por un nulo). Todos los arreglos tienen el 0 como primer elemento. Hay que tener muy presente no rebasar el último índice. La cantidad de memoria requerida para guardar un arreglo está directamente relacionada con su tipo y su tamaño.

SINTAXIS:

```
tipo nombre_arreglo [nº elementos];

tipo nombre_arreglo [nº elementos]={valor1,valor2,valorN};

tipo nombre_arreglo[]={valor1,valor2,valorN};
```

INICIALIZACIÓN DE UN ELEMENTO:

```
nombre_arreglo[indice]=valor;
```

UTILIZACIÓN DE ELEMENTOS:

```
nombre_arreglo[indice];
```

Ejercicio 5.1.

Reserva 100 elementos enteros, los inicializa todos y muestra el 5º elemento.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
   int x[100];
   int cont;

   for(cont=0;cont<100;cont++)
         x[cont]=cont;

   printf("%d",x[4]);
   getch();
}
```

Ejercicio 5.2.

Veamos un ejercicio con arreglos utilizando números, el programa sumará, restará, multiplicará, dividirá los tres elementos de un arreglo denominado datos, y almacenará los resultados en un segundo arreglo denominado resul.

```
#include<stdio.h>
#include<conio.h>

void main(void)
{
      static int datos[3]={10,6,20};

      static int resul[5]={0,0,0,0,0};

      resul[0]=datos[0]+datos[2];
      resul[1]=datos[0]-datos[1];
      resul[2]=datos[0]*datos[2];
      resul[3]=datos[2]/datos[1];
      resul[4]=datos[0]%datos[1];

      printf("\nSuma: %d",resul[0]);
      printf("\nResta: %d",resul[1]);
      printf("\nMultiplicacion: %d",resul[2]);
      printf("\nDivision: %d",resul[3]);
      printf("\nResiduo: %d",resul[4]);
}
```

Ejercicio 5.3.

El siguiente ejercicio carga un arreglo de enteros con números desde el cero hasta el 99 y los imprime en pantalla.

```
#include <stdio.h>
int main(void)
{
      int x[100]; /* declara un arreglo de 100-integer */
      int t;
      /* carga x con valores de 0 hasta 99 */
      for(t=0; t<100; ++t)
            x[t] = t;
      /* despliega el contenido de x */
      for(t=0; t<100; ++t)
            printf("%d ", x[t]);
      return 0;
```

```
}
```

5.2. Arreglos multidimensionales

C admite arreglos multidimensionales, la forma más sencilla es la de los arreglos bidimensionales, éstos son esencialmente un arreglo de arreglos unidimensionales, se almacenan en matrices fila-columna, el primer índice muestra la fila y el segundo la columna. Esto significa que el índice más a la derecha cambia más rápido que el de más a la izquierda cuando accedemos a los elementos.

SINTAXIS:

```
tipo nombre_arreglo[fila][columna];

tipo
nomb_arreglo[fil][col]={{v1,v2,vN},{v1,v2,vN},{vN}};

tipo nomb_arreglo[][]={{v1,v2,vN},{v1,v2,vN},{vN}};
```

num [fila] [columna]

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

INICIALIZACIÓN DE UN ELEMENTO:

```
nombre_arreglo[indice_fila][indice_columna]=valor;
```

UTILIZACIÓN DE UN ELEMENTO:

```
nombre_arreglo[indice_fila][indice_columna];
```

Para conocer el tamaño que tiene un *array* bidimensional tenemos que multiplicar las filas por las columnas, por el número de bytes que ocupa en memoria el tipo del *array*. Es exactamente igual que con los *array* unidimensionales lo único que se añade son las columnas.

```
filas * columnas * bytes_del_tipo
```

Ejercicio 5.4.

Ejemplo de un arreglo de dos dimensiones.

El programa utiliza un arreglo de dos dimensiones y lo llena con el valor de 1

```
# include <stdio.h>
#define R 5
#define C 5
main()
{
      int matriz[R][C];
      int i,j;
      for(i=0;i<R;i++)
            for(j=0;j<C;j++)
            matriz[i][j] = 1;
      for(i=0;i<R;i++)
            for(j=0;j<C;j++)
                  printf("%d\n",matriz[i][j]);
            for(i=0;i<R;i++)
            {
                  printf("NUMERO DE RENGLON: %d\n",i);
                  for(j=0;j<C;j++)
                        printf("%d ",matriz[i][j]);
                  printf("\n\n");
            }
      printf("Fin del programa");
      return(0);
}
```

Ejercicio 5.5.

El siguiente ejercicio suma dos matrices.

```
#include <stdio.h>
void main(void)
{
      int matrix1[2][2];
      int matrix2[2][2];
      int res[2][2];

      printf("Introduzca los cuatro valores de la matriz 1:
");
      scanf("%d %d %d %d",
      &matrix1[0][0],&matrix1[0][1],&matrix1[1][0],&matrix1[1
][1]);

      printf("Introduzca los cuatro valores de la matriz 2:
");
      scanf("%d %d %d %d",
&matrix2[0][0],&matrix2[0][1],&matrix2[1][0],&matrix2[1][1]);

      res[0][0] = matrix1[0][0] + matrix2[0][0];
      res[0][1] = matrix1[0][1] + matrix2[0][1];
      res[1][0] = matrix1[1][0] + matrix2[1][0];
      res[1][1] = matrix1[1][1] + matrix2[1][1];

      printf("%d %d\n",res[0][0],res[0][1]);
      printf("%d %d\n",res[1][0],res[1][1]);
}
```

5.3. Arreglos y cadenas

El uso más común de los arreglos unidimensionales es su utilización con **cadenas**, conjunto de caracteres terminados por el carácter nulo ('\0'). Por tanto cuando se quiera declarar un arreglo de caracteres se pondrá siempre una posición más. No es necesario añadir explícitamente el carácter nulo, el compilador lo hace automáticamente.

SINTAXIS:

```
char nombre[tamaño];

char nombre[tamaño]="cadena";

char nombre[]="cadena";

char nombre[]='c';
```

Ejercicio 5.6.

Realiza un programa en C en el que se representen los tres estados del agua con una cadena de caracteres. El arreglo agua_estado1 debe ser inicializado carácter por carácter con el operador de asignación, ejemplo: agua_estado1[0] = 'g'; el arreglo agua_estado2 será inicializado utilizando la función **scanf**; y el arreglo agua_estado3 deberá ser inicializado de una sola, utiliza la palabra **static**.

```
/* Ejemplo del uso de arrays y cadenas de caracteres */

#include <stdio.h>

main()

{
      char agua_estado1[4],                     /* gas */
      agua_estado2[7];                      /* solido */
      static char agua_estado3[8] = "liquido";  /* liquido */
      agua_estado1[0] = 'g';
      agua_estado1[1] = 'a';
      agua_estado1[2] = 's';
      agua_estado1[3] = '\0';
      printf("\n\tPor favor introduzca el estado del agua ->
solido ");
      scanf("%s",agua_estado2);
      printf("%s\n",agua_estado1);
      printf("%s\n",agua_estado2);
      printf("%s\n",agua_estado3);
      return(0);
}
```

5.4. Estructuras

C proporciona formas diferentes de creación de tipos de datos propios. Uno de ellos es la agrupación de variables bajo un mismo nombre, otra es permitir que la

misma parte de memoria sea definida como dos o más tipos diferentes de variables y también crear una lista de constantes enteras con nombre.

Una **estructura** es una colección de variables que se referencia bajo un único nombre, proporcionando un medio eficaz de mantener junta una información relacionada. Las variables que componen la estructura se llaman miembros de la estructura y están relacionadas lógicamente con las otras. Otra característica es el ahorro de memoria y evitar declarar variables que técnicamente realizan las mismas funciones.

SINTAXIS:

```
struct nombre{
 var1;
     var2;
     varN;
};
 .
 .
struct nombre etiqueta1,etiquetaN;
```

Los miembros individuales de la estructura se referencian utilizando la etiqueta de la estructura, el operador punto(.) y la variable a la que se hace referencia. Los miembros de la estructura deben ser inicializados fuera de ella, si se hace en el interior da error de compilación.

```
etiqueta.variable;
```

Ejercicio 5.7.

El siguiente programa almacena información acerca de automóviles, debido a que se maneja información de distintos tipos, seria imposible manejarla en un arreglo, por lo que es necesario utilizar una estructura.

El programa le pide al usuario que introduzca los datos necesarios para almacenar información de automóviles, cuando el usuario termina la captura, debe presionar las teclas ctrl.+z.

```
/* programa en C que muestra la forma de crear una estructura
*/

# include <stdio.h>

struct coche {
   char fabricante[15];
   char modelo[15];
   char matricula[20];
   int antiguedad;
   long int kilometraje;
   float precio_nuevo;
   } miauto;

main()
{
   printf("Introduzca el fabricante del coche.\n");
   gets(miauto.fabricante);
   printf("Introduzca el modelo.\n");
   gets(miauto.modelo);
   printf("Introduzca la matricula.\n");
   gets(miauto.matricula);
   printf("Introduzca la antiguedad.\n");
   scanf("%d",&miauto.antiguedad);
   printf("Introduzca el kilometraje.\n");
   scanf("%ld",&miauto.kilometraje);
   printf("Introduzca el precio.\n");
   scanf("%f",&miauto.precio_nuevo);
   getchar(); /*vacia la memoria intermedia del teclado*/

   printf("\n\n\n");
   printf("Un %s %s con %d años de antiguedad con número de
matricula
#%s\n",miauto.fabricante,miauto.modelo,miauto.antiguedad,miau
to.matricula);
   printf("actualmente con %ld
kilómetros",miauto.kilometraje);
   printf(" y que fue comprado por
$%5.2f\n",miauto.precio_nuevo);
   return(0);
}
```

Ejercicio 5.8.

Realiza un programa que utilice la estructura anterior, pero que se encuentre dentro de un ciclo *for* para que el usuario pueda determinar cuántos autos va a introducir.

```
/* programa en C que muestra la forma de crear una estructura
*/

# include <stdio.h>

int i,j;
struct coche {
   char fabricante[15];
   char modelo[15];
   char matricula[20];
   int antiguedad;
   long int kilometraje;
   float precio_nuevo;
   } miauto;

main()
{
   printf("Introduce cuantos autos vas a capturar:\n");
   scanf("%d",&j);
   flushall(); /*vacia la memoria intermedia del teclado */
   for(i=1;i<=j;i++)
   {
      printf("Introduzca el fabricante del coche.\n");
      gets(miauto.fabricante);
      printf("Introduzca el modelo.\n");
      gets(miauto.modelo);
      printf("Introduzca la matricula.\n");
      gets(miauto.matricula);
      printf("Introduzca la antiguedad.\n");
      scanf("%d",&miauto.antiguedad);
      printf("Introduzca el kilometraje.\n");
      scanf("%ld",&miauto.kilometraje);
      printf("Introduzca el precio.\n");
      scanf("%f",&miauto.precio_nuevo);
      getchar(); /*vacia la memoria intermedia del teclado*/

      printf("\n\n\n");
      printf("Un %s %s con %d años de antiguedad con número
de matricula
```

```
#%s\n",miauto.fabricante,miauto.modelo,miauto.antiguedad,miau
to.matricula);
      printf("actualmente con %ld
kilómetros",miauto.kilometraje);
      printf(" y que fue comprado por
$%5.2f\n",miauto.precio_nuevo);
   flushall();
   }
   return(0);
}
```

Ejercicio 5.9.

Veamos ahora un ejemplo con arreglos, cadenas y estructuras.

```
/* Inicialización y manejo de "arreglos", cadenas y
estructuras.*/
# include <stdio.h>
void main()
{
int i, j;
static int enteros [5] = { 3, 7, 1, 5, 2 };
static char cadena1 [16] = "cadena";
static char cadena2 [16] = { 'c','a','d','e','n','a','\0' };
static int a[2][5] = {
{ 1, 22, 333, 4444, 55555 },
{ 5, 4, 3, 2, 1 }
};
static int b[2][5] = { 1,22,333,4444,55555,5,4,3,2,1 };
static char *c = "cadena";
static struct {
int i;
float x;
} sta = { 1, 3.1415e4}, stb = { 2, 1.5e4 };
static struct {
char c;
int i;
float s;
} st [2][3] = {
{{ 'a', 1, 3e3 }, { 'b', 2, 4e2 }, { 'c', 3, 5e3 }},
{ { 'd', 4, 6e2 },}
};
printf ("enteros:\n");
for ( i=0; i<5; ++i ) printf ("%d ", enteros[i]);
printf ("\n\n");
printf ("cadena1:\n");
```

```
printf ("%s\n\n", cadena1);
printf ("cadena2:\n");
printf ("%s\n\n", cadena2);
printf ("a:\n");
for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d ",
a[i][j]);
printf("\n\n");
printf ("b:\n");
for (i=0; i<2; ++i) for (j=0; j<5; ++j) printf ("%d ",
b[i][j]);
printf ("\n\n");
printf ("c:\n");
printf ("%s\n\n", c);
printf ("sta:\n");
printf ("%d %f \n\n", sta.i, sta.x);
printf ("st:\n");
for (i=0; i<2; ++i) for (j=0; j<3; ++j)
printf ("%c %d %f\n", st[i][j].c, st[i][j].i, st[i][j].s);
}
```

La salida del anterior programa es la siguiente:

```
enteros:
3 7 1 5 2
cadena1:
cadena
cadena2:
cadena
a:
1 22 333 4444 55555 5 4 3 2 1
b:
1 22 333 4444 55555 5 4 3 2 1
c:
cadena
sta:
1 31415.000000
st:
a 1 3000.000000
b 2 400.000000
c 3 5000.000000
d 4 600.000000
0 0.000000
0 0.000000
```

CONCLUSIONES

Los arreglos permiten la utilización de varios valores a través del nombre de una variable. Un uso muy común de los arreglos, es la gestión de cadenas, C permite dicha manipulación de manera muy ágil y precisa.

Por otro lado las estructuras permiten la gestión de varios tipos de datos, lo que permite almacenar información diversa.

Bibliografía del tema 5

www.monografias.com

www.lawebdelprogramador.com

Actividades de aprendizaje

A.5.1. Realiza un programa que busque en un arreglo de 100 números, un número que el usuario determine.

A.5.2. Realiza un programa que simule la operación "*push*" de una pila, utilizando arreglos.

A.5.3. Realiza un programa que simule la operación "*pop*" de una pila, utilizando arreglos.

A.5.4. Realiza un programa que determine el tamaño de un arreglo bidimensional.

A.5.5. Realiza un programa que ordene 10 números almacenados en un arreglo.

Cuestionario de autoevaluación

1. ¿Qué es un arreglo?
2. ¿Qué es un vector?
3. ¿Qué es el carácter nulo?
4. ¿Qué es una matriz?
5. ¿Qué es una estructura?
6. ¿Qué es una enumeración?
7. ¿Qué significa la palabra NULL?

8. ¿Qué es una cadena?
9. ¿Qué es una etiqueta dentro de una estructura?
10. ¿Qué es un arreglo multidimensional?

Examen de autoevaluación

1. El primer elemento de un arreglo es el número:
    a. 1
    b. 0
    c. -1
    d. NULL

2. Un vector es sinónimo de:
    a. Arreglo unidimensional
    b. Campo
    c. Registro
    d. Índice

3. Para acceder al contenido de un arreglo bidimensional se necesita:
    a. Usar un índice
    b. Usar dos índices
    c. Usar tres índices
    d. Usar cuatro índices

4. Un arreglo puede tener:
    a. Más de dos dimensiones
    b. Solo una dimensión
    c. Solo dos dimensiones
    d. Cero dimensiones

5. Un arreglo puede manejar:
    a. Sólo un tipo de dato
    b. Sólo caracteres
    c. Sólo enteros
    d. Sólo cadenas

6. Una cadena termina con:
    a. '\r'
    b. '\t'
    c. '\n'
    d. '\0'

7. Para inicializar un carácter en un arreglo se usa:
    a. La comilla simple
    b. La comilla doble
    c. No se usan comillas
    d. Dos comillas simples

8. Para inicializar un número en un arreglo se usa un:
    a. Número con la comilla simple
    b. Número la comilla doble
    c. Número sin comillas
    d. Número con dos comillas simples

9. Una estructura puede manejar:
    a. Un solo tipo de dato
    b. Sólo cadenas

c. Cadenas y caracteres
d. Varios tipos de datos

10. Los elementos de una estructura se referencian a través de:
    a. Un punto.
    b. Una coma ,
    c. Una flecha ->
    d. Un paréntesis ()

TEMA 6. MANEJO DE APUNTADORES

Objetivo particular

Que el alumno comprenda el uso de los apuntadores en combinación con los arreglos, y las funciones, además de comprender el uso de la memoria dinámica.

Temario detallado

6.1. Introducción a los apuntadores
6.2. Apuntadores y arreglos
6.3. Apuntadores y estructuras
6.4. Apuntadores y funciones
6.5. Manejo dinámico de memoria

Introducción

El uso de los apuntadores permite el manejo de la memoria de una manera más eficiente, los apuntadores son muy utilizados para el almacenamiento de memoria de manera dinámica.

6.1. Introducción a los apuntadores

Un **apuntador** o **puntero** es una variable que contiene una dirección de memoria. Esa dirección es la posición de un objeto (normalmente una variable) en memoria. Si una variable va a contener un puntero, entonces tiene que declararse como tal. Cuando se declara un puntero que no apunte a ninguna posición válida ha de ser asignado a un valor nulo (un cero.

SINTAXIS:

```
tipo *nombre;
```

OPERADORES: El **&** devuelve la dirección de memoria de su operando. El * devuelve el valor de la variable localizada en la dirección que sigue. Pone en **m** la dirección de memoria de la variable cuenta. La dirección no tiene nada que ver con el valor de cuenta. En la segunda línea pone el valor de cuenta en **q**.

```
m= &cuenta;

q=*m;
```

Ejercicio 6.1.

El siguiente programa obtiene la dirección de memoria de una variable introducida por el usuario.

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
      int *p;
      int valor;

      printf("Introducir valor: ");
      scanf("%d",&valor);

      p=&valor;

      printf("Direccion de memoria de valor es: %p\n",p);
      printf("El valor de la variable es: %d",*p);
      getch();
}
```

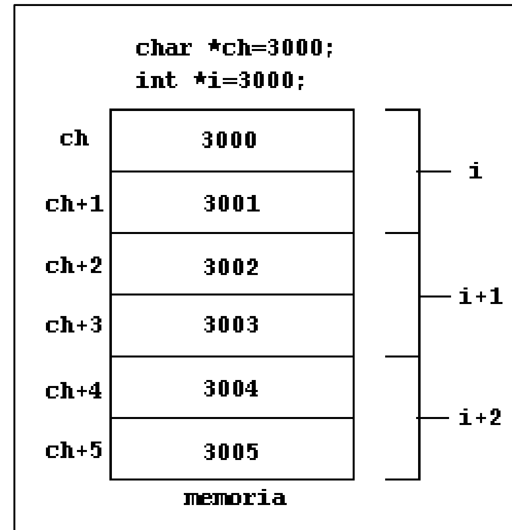
El siguiente ejercicio utiliza dos apuntadores, en el apuntador p1 se guara la dirección de memoria de la variable x y en la variable p2 se asigna el contenido del apuntador p1, por ultimo se imprime el contenido del apuntador p2.

Ejercicio 6.2.

```
#include <stdio.h>
void main(void)
{
      int x;
      int *p1, *p2;
      p1=&x;
      p2=p1;
      printf("%p",p2);
}
```

ARITMÉTICA: Las dos operaciones aritméticas que se pueden usar como punteros son la suma y la resta. Cuando se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cada vez que se decrementa, apunta a la posición del elemento anterior.

```
p1++;
p1--;
p1+12;
```



COMPARACIÓN: Se pueden comparar dos punteros en una expresión relacional. Generalmente la comparación de punteros se utiliza cuando dos o más punteros apuntan a un objeto común.

```
if(p<q) printf ("p apunta a menor memoria que q");
```

6.2. Apuntadores y arreglos

Los apuntadores pueden estructurarse en arreglos como cualquier otro tipo de datos. Hay que indicar el tipo y el número de elementos. Su utilización posterior es igual a los arreglos que hemos visto anteriormente, con la diferencia de que se asignan **direcciones de memoria**.

DECLARACIÓN:

```
tipo *nombre[nº elementos];
```

ASIGNACIÓN:

```
nombre_arreglo[indice]=&variable;
```

UTILIZAR ELEMENTOS:

```
*nombre_arreglo[indice];
```

Ejercicio 6.3.

El siguiente programa utiliza un arreglo de apuntadores. El usuario debe introducir un número de día, y dependiendo de la selección, el programa el día adecuado.

```
#include <stdio.h>
#include <conio.h>

void dias(int n);

void main(void)
{
   int num;
   printf("Introducir nº de Dia: ");
   scanf("%d",&num);
   dias(num);
   getch();
}

void dias(int n)
{
   char *dia[]={"Nº de dia no Valido",
   "Lunes",
   "Martes",
   "Miercoles",
   "Jueves",
   "viernes",
   "Sabado",
   "Domingo"};

   printf("%s",dia[n]);
}
```

6.3. Apuntadores y estructuras

C permite punteros a estructuras igual que permite punteros a cualquier otro tipo de variables. El uso principal de este tipo de punteros es **pasar estructuras a funciones**. Si tenemos que pasar toda la estructura el tiempo de ejecución puede hacerse eterno, utilizando punteros sólo se pasa la dirección de la estructura. Los punteros a estructuras se declaran poniendo ***** delante de la etiqueta de la estructura.

SINTAXIS:

```
struct nombre_estructura etiqueta;
struct nombre_estructura *nombre_puntero;
```

Para encontrar la dirección de una etiqueta de estructura, se coloca el operador **&** antes del nombre de la etiqueta. Para acceder a los miembros de una estructura usando un puntero usaremos el operador flecha (**->**).

Ejercicio 6.4.

El siguiente programa utiliza una estructura con apuntadores para almacenar los datos de un empleado, se utliza una estructura con tres elementos: nombre, dirección y telefono, se utilza además una función para mostrar el nombre del empleado en pantalla.

```
#include <stdio.h>
#include <string.h>
typedef struct
{
      char nombre[50];
      char direccion[50];
      long int telefono; }empleado

      void mostrar_nombre(empleado *x);
```

```
int main(void)
{
        empleado mi_empleado;
        empleado * ptr;
        ptr = &mi_empleado;     /* sintaxis para
modificar un miembro de una estructura a través de un
apuntador */
        ptr->telefono = 5632332;
        strcpy(ptr->nombre, "jonas");
        mostrar_nombre(ptr);
        return 0;
}
void mostrar_nombre(empleado *x)
{
        /* se debe de usar -> para referenciar a los
miembros de la estructura */
        printf("nombre del empleado x %s \n", x->nombre);
}
```

La estructura se pasa por referencia:

```
mostrar_nombre(ptr);
```

Cuando se ejecuta la anterior instrucción es como si se "ejecutara" la instrucción:

```
empleado *x = ptr;
```

6.4. Apuntadores y funciones

En la unidad de funciones, se utilizo el algoritmo de la burbuja para ordenar elementos de un mismo tipo.

Pero si nuestro objetivo es hacer de nuestra rutina de ordenación independiente del tipo de datos, una manera de lograrlo es usar apuntadores sin tipo (void) para que apunten a los datos en lugar de usar el tipo de datos enteros.

Ejercicio 6.4.

```
#include <stdio.h>
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
void bubble(int *p, int N);
int compare(int *m, int *n);
int main(void)
{
	int i;
	putchar('\n');
	for (i = 0; i < 10; i++)
	{
		printf("%d ", arr[i]);
	}
	bubble(arr,10);
	putchar('\n');
	for (i = 0; i < 10; i++)
	{
		printf("%d ", arr[i]);
	}
	return 0;
}

void bubble(int *p, int N)
{
	int i, j, t;
	for (i = N-1; i >= 0; i--)
	{
		for (j = 1; j <= i; j++)
		{
			if (compare(&p[j-1], &p[j]))
			{
				t = p[j-1];
				p[j-1] = p[j];
				p[j] = t;
			}
		}
	}
}

int compare(int *m, int *n)
{
	return (*m > *n);
}
```

Estamos pasando un apuntador a un entero (o a un arreglo de enteros) a **bubble()**.

Y desde dentro de bubble estamos pasando apuntadores a los elementos que queremos comparar del arreglo a nuestra función de comparación. Y por supuesto estamos desreferenciando estos apuntadores en nuestra función **compare()** de modo que se haga la comparación real entre elementos. Nuestro siguiente paso será convertir los apuntadores en **bubble()** a apuntadores sin tipo de tal modo que la función se vuelva más insensible al tipo de datos a ordenar.

6.5. Manejo dinámico de memoria

La asignación dinámica es la forma en que un programa puede obtener memoria mientras se está ejecutando, debido a que hay ocasiones en que se necesita usar cantidades de memoria variable.

La memoria se obtiene del **montón** (región de la memoria libre que queda entre el programa y la pila). El tamaño del **heap** (montón) es desconocido pero contiene gran cantidad de memoria. El sistema de asignación dinámica está compuesto por las funciones **malloc** que asignan memoria a un puntero y **free** que libera la memoria asignada. Ambas funciones necesitan el archivo de cabecera **stdlib.h**.

Al llamar a **malloc**, se asigna un bloque contiguo de almacenamiento al objeto especificado, devolviendo un puntero al comienzo del bloque. La función **free** libera memoria previamente asignada dentro del heap, permitiendo que ésta sea reasignada cuando sea necesario.

El argumento pasado a **malloc** es un entero sin signo que representa el número de bytes de almacenamiento requeridos. Si el almacenamiento está disponible, **malloc** devuelve un **void ***, que se puede transformar en el puntero de tipo deseado. El concepto de punteros **void** se introdujo en el C ANSI estándar, y significa un puntero de tipo desconocido, o un puntero genérico.

SINTAXIS:

```
puntero=malloc(numero_de_bytes);

free(puntero);
```

El siguiente segmento de código asigna almacenamiento suficiente para 200 valores flota:

```
float *apun_float;
int num_floats = 200;
apun_float = malloc(num_floats * sizeof(flota));
```

La función **malloc** se ha utilizado para obtener almacenamiento suficiente para 200 por el tamaño actual de un **float**. Cuando el bloque ya no es necesario, se libera por medio de:

```
free(apun_float);
```

Ejercicio 6.4.

Después de que el programa define la variable **int *block_mem**, se llama a la función **malloc**, para asignar un espacio de memoria suficiente para almacenar una variable de tipo **int**.

Para obtener el tamaño adecuado se usa la función **sizeof.** Si la asignación es exitosa se manda un mensaje en pantalla, de la contrario se indica que la memoria es insuficiente,

Por último se libera la memoria ocupada a través de la función **free**.

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 256
```

```
main()
{
      int *block_mem;
      block_mem=(int *)malloc(MAX * sizeof(int));
      if(block_mem == NULL)
            printf("Memoria insuficiente\n");
      else {
            printf("Memoria asignada\n");
            free(block_mem);
      }
return(0);
}
```

CONCLUSIONES

La principal ventaja de los apuntadores la utilización de memoria dinámica. A diferencia de los arreglos o las estructuras, los apuntadores permiten en uso de un conjunto de datos que puede aumentar o disminuir de tamaño durante la ejecución del programa.

Bibliografía del tema 6

www.lawebdelprogramador.com

wiki.lidsol.net

Actividades de aprendizaje

A.6.1. Realiza un programa que simule la operación “*push*” de una pila utilizando apuntadores.

A.6.2. Realizar un programa que simule la operación “*pop*” de una pila utilizando apuntadores.

A.6.3. Realiza un programa que muestre la dirección de memoria de una variable.

A.6.4. Realiza un programa que simule una cola, utilizando apuntadores.

A.6.5. Realiza un programa que use memoria dinámica para crear una lista simplemente enlazada.

Cuestionario de autoevaluación

1. ¿Qué es un apuntador?
2. ¿Cuál es la utilidad de un apuntador?
3. ¿Un apuntador puede apuntar a un apuntador?
4. ¿Qué es una dirección de memoria?
5. ¿Qué significa el término de memoria dinámica?
6. ¿Para que se usa el símbolo de asterisco (*****)?
7. ¿Para que se usa el símbolo de ampersand (**&**)?
8. ¿Cuál es la utilidad de los arreglos de apuntadores?
9. ¿Cuál es la utilidad de las funciones en combinación con los apuntadores?
10. ¿Qué es el **heap**?

Examen de autoevaluación

1. Un apuntador es:
   a. Una variable que contiene una dirección de memoria
   b. Una dirección de memoria
   c. Una variable
   d. El signo de *

2. Para acceder a la dirección de memoria se usa:
   a. ->
   b. &
   c. >>
   d. >

3. Para acceder al contenido de una variable se usa:
   a. <-
   b. &
   c. *
   d. <<

4. El operador que se usa para imprimir una apuntador es:
    a. *
    b. &
    c. %
    d. %i

5. La función de *malloc* es la de:
    a. Asignar un espacio de memoria
    b. Obtener una dirección de memoria
    c. Definir un tipo de dato
    d. Definir un apuntador

6. La función de free es la de:
    a. Obtener una dirección de memoria
    b. Liberar memoria
    c. Obtener una dirección de memoria disponible
    d. Definir un tipo generico

7. La función de sizeof es la de:
    a. Determinar el tamaño de un tipo de dato
    b. Determinar el tamaño de un apuntador
    c. Determinar el tamaño de un arreglo
    d. Determinar el tamaño de una estructura

8. Las funciones malloc y free se encuentran en la biblioteca:
    a. stdlib.h
    b. sodio.h
    c. string.h
    d. graph.h

9. Para acceder a un elemento de una estructura usando apuntadores se usa:
    a. <-
    b. ->
    c. >>
    d. <<

10. Los tipos de los apuntadores:
    a. Sólo son tipos enteros
    b. Sólo son de tipo carácter
    c. Sólo son de tipo flotante
    d. Pueden no tener n tipo definido

TEMA 7. ARCHIVOS

Objetivo particular

La presente unidad tiene por objetivo que el alumno conozca la gestión de archivos de texto, a través del lenguaje C.

Temario Detallado

7.1. Creación de un archivo
7.2. Consulta de un archivo
7.3. Modificación de un archivo
7.4. Eliminación de un archivo

Introducción

La gestión de los archivos, es una tarea que realizan prácticamente todos los sistemas de información, y el lenguaje C cuenta con una rica gama de funciones para la gestión de archivos, en esta última unidad se verán varios ejemplos para la creación, consulta e incluso eliminación de archivos.

La biblioteca de rutinas estándar de Entrada/Salida en C permite leer y escribir datos a/o desde archivos y dispositivos. El lenguaje C no incluye ninguna estructura de archivo predefinida. En su lugar, todos los datos se tratan como un flujo o secuencia (*stream*) de bytes. Hay tres tipos básicos de funciones E/S: orientadas a flujo, a consola y puerto, y funciones de bajo nivel.

Todas las funciones de E/S orientadas a flujo tratan los archivos de datos o los datos simples como un flujo de caracteres individuales. Si seleccionamos la función de flujo adecuada, nuestra aplicación podrá procesar datos de cualquier tamaño o formato requerido, desde caracteres simples a grandes y complicadas estructuras de datos.

Técnicamente, cuando un programa utiliza una función de flujo para abrir un archivo para E/S, éste se asocia con una estructura de tipo **FILE** (predefinida en

stdio.h) que contiene información básica acerca del archivo. Una vez que el flujo está abierto se devuelve un puntero a una estructura de tipo FILE, llamado a veces apuntador de flujo o flujo, se utiliza para referirse al archivo en todas las E/S posteriores.

Todas las **funciones de flujo** E/S proporcionan entrada/salida con memoria intermedia (**buffer**), formateada o no formateada. Un flujo proporciona una memoria intermedia para toda la información que entre al flujo o sea enviada a través de él. La E/S en disco es lenta, pero el uso de memorias intermedias puede mejorar nuestra aplicación. En lugar de introducir en el flujo de datos un carácter o una estructura, las funciones de E/S de flujo acceden a un bloque de datos a la vez. Cuando la aplicación necesite procesar la entrada, sólo tendrá que acceder a la memoria intermedia, que es mucho más rápida. Cuando la memoria intermedia se descargue, se leerá otro bloque del disco.

Para la salida de datos el proceso es el inverso. En lugar de escribir físicamente los datos, cuando se ejecuta cada instrucción de salida, las funciones de E/S orientadas a flujo sitúan todos los datos de salida en la memoria intermedia. Cuando ésta se llene, se escribirán los datos en el disco.

APERTURA DE UN FLUJO:

```
fich = fopen(muestra.dat”,”r”);
```

FUNCIONES USUALES EN ARCHIVOS

FUNCIÓN DESCRIPCIÓN

```
fopen()     abre un flujo
fclose()    cierra un flujo
putc()      escribe un carácter un flujo
getc()      lee un carácter desde un flujo
fseek()     salta al byte especificado en un flujo
fprintf()   escribe en disco
fscanf()    lee de disco
```

```
fputs()     lee una cadena
fgets()     lee una cadena
feof        devuelve verdadero si se llega al final: EOF
remove()    borra un archivo
```

MODOS DE ABRIR UN ARCHIVO EN C

Los archivos pueden abrirse en modo texto o en modo binario. En modo texto, la mayoría de los compiladores de C traducen las secuencias de retorno de carro/cambio de línea en caracteres de avance de entrada. En la salida se produce el proceso opuesto. Sin embargo, los archivos binarios no pasan por tales traducciones.

```
w     crea un archivo de texto para escribir en el
r     abre un archivo para leer
a     añade texto a un archivo
wb    crea un archivo binario para escribir
rb    abre un archivo binario para leer
ab    añade texto a un archivo binario
w+    crea un archivo para leer y escribir
r+    abre un archivo para leer y escribir
a+    abre o crea un archivo de texto para leer y escribir
r+b   abre un archivo binario para leer y escribir
w+b   crea un archivo binario para leer y escribir
a+b   abre un archivo binario para leer y escribir
rt    abre un archivo de texto para leer
wt    crea un archivo de texto para escribir
at    añade un archivo de texto
w+t   crea un archivo de texto para leer y escribir
r+t   abre un archivo de texto para leer y escribir
a+t   abre o crea un archivo de texto para leer y escribir
```

7.1. Creación de un archivo

Es la primera operación que sufrirá el archivo de datos. Implica la elección de un entorno descriptivo que permita un ágil, rápido y eficaz tratamiento del archivo.

Un archivo puede ser creado por primera vez en un soporte, proceder de otro previamente existente en el mismo o diferente soporte, ser el resultado de un cálculo o ambas cosas a la vez.

Ejercicio 7.1.

El siguiente ejercicio crea un archivo de texto de nombre “**prueba1.txt**”, cabe destacar que la extensión txt no influye en el contenido del archivo, se le puede dar otra extensión, o incluso no darle extensión. Después de creado el archivo se introduce el texto “**Texto grabado en el archivo prueba1.txt**”.

En caso de que no se pueda crear el archivo, el programa manda un mensaje indicándolo. Por último se cierra el archivo.

```
# include <stdio.h>

main()
{
   FILE *in;

   if( (in=fopen("prueba1.txt","w")) != NULL)
   {
      fputs("Texto grabado en el archivo prueba1.txt",in);
      fclose(in);
   }
   else
      printf("No se puede crear el archivo prueba1.txt\n");
   return(0);
}
```

7.2. Consulta de un archivo

Es la operación que permite al usuario acceder al archivo de datos para conocer el contenido de uno, varios o todos los registros.

Ejercicio 7.2.

Veamos un programa en C que consulta el contenido de un archivo y cuenta sus caracteres. El programa abre el archivo "**prueba1.txt**", en modo de lectura, y mientras no se alcance el fin del archivo (**EOF**), un carácter será almacenado en la variable **ch**, a través de la función **getc**, después a través de la función **putc**, se muestra el carácter leído en pantalla. Por último se cierra el archivo.

```
/* Consulta el contenido de un archivo */
# include <stdio.h>
# include <conio.h>

main()
{
   FILE *in;
   int ch,cuenta=0;

   /* Extrae los datos del archivo creado */
   if( (in=fopen("prueba1.txt","r") )!=NULL)
   {
      while ( ( ch=getc(in) ) != EOF)
      {
         putch(ch);
         cuenta=cuenta+1;
        }
      fclose(in);
   }
   else
      printf("No se encuentra el archivo %s.\n",in);

   printf("La cantidad de caracteres es de: %d\n",cuenta);
   return(0);
}
```

Ejercicio 7.3.
El programa pide el nombre del archivo y lo guarda en el arreglo **archivo**, de la misma manera pide un texto que almacena en el arreglo línea, posteriormente utiliza el contenido de dichos arreglos para darle un nombre al archivo, y escribir sobre ellos el texto que haya escrito el usuario. Después muestra el contenido del archivo. Por último cierra el archivo.

```
/* Consulta el contenido de un archivo */
# include <stdio.h>
# include <conio.h>

main()
{
   FILE *in;
   int ch;
   char archivo[81];
   char linea[81];

   printf( "Introduzca el nombre del archivo: " );
   gets( archivo );
   printf( "\nIntroduzca el texto: " );
   gets( linea );

   if( (in=fopen(archivo,"w")) != NULL)
   {
      fputs(linea,in);
      fclose(in);
   }
   else
      printf("No se puede crear el archivo %s\n",archivo);

   if( (in=fopen(archivo,"r") )!=NULL)
   {
      printf("Contenido del archivo %s: \n",archivo);
      while ( ( ch=getc(in) ) != EOF)
      putch(ch);
      fclose(in);
   }
   else
      printf("No se encuentra el archivo %s",archivo);
      getch();
      return(0);
}
```

7.4. Modificación de un archivo

La actualización de un archivo, es la modificación del contenido de un archivo, el contenido puede ser sobre escrito, se le pueden agregar nuevos registros, o modificar una parte del archivo.

Ejercicio 7.3.
El siguiente programa abre el archivo prueba.txt en modo "**append**", o sea que se pueden agregar caracteres, y no se elimina el contenido del archivo.
El programa le pide una cadena al usuario, la cual posteriormente es agregada al archivo. Por último cierra el archivo.

```
/* programa que inserta varios registros en un archivo */

# include <stdio.h>

main()
{
   FILE *in;
   char linea[81];

   if( (in=fopen("prueba.txt","a")) != NULL)
   {

     printf( "Introduzca una cadena: " );
     gets( linea );
     printf( "La linea introducida fue: %s\n", linea );
     fputs(linea,in);
     fclose(in);
   }
   else
      printf("No se puede crear el archivo prueba.txt\n");

      return(0);
}
```

7.4. Eliminación de un archivo

Como su nombre lo indica, es eliminar del soporte físico un archivo.

Ejercicio 7.5.

El siguiente programa elimina un archivo que el usuario indique, la función que elimina el archivo es **remove**. En caso de que no se pueda eliminar el archivo se manda un mensaje en pantalla indicandolo.

```
#include <stdio.h>
char archivo[40];

void main( void )
{

   printf("Introduzca el nombre del archivo que desea eliminar:
");
   gets(archivo);
   printf("El archivo es: %s",archivo);

   if( remove(archivo) == -1 )
      perror( "\nNo se pudo eliminar el archivo");
   else
      printf( "\nSe elimino %s\n",archivo );
}
```

CONCLUSIONES

La mayoría de los sistemas de información almacenan información. Por lo que el uso de archivos es una tarea común. Cabe destacar que sistemas operativos como Unix utilizan archivos de texto, para la gestión del sistema operativo y que la mayoría de los manejadores de bases de datos, fueron desarrollados en lenguaje C.

Bibliografía del tema 7

www.lawebdelprogramador.com

Actividades de aprendizaje

A.7.1. Investiga la diferencia entre un archivo de texto y un archivo binario.

A.7.2. Investiga como se realiza la gestión de archivo en el lenguaje C++.

A.7.3. Investiga como se realiza la gestión de archivo en el lenguaje Java.

A.7.4. Realiza un programa que ordene 10 números de menor a mayor, que se encuentran en un archivo. Utiliza el archivo de la burbuja.

A.7.5. Realiza un programa que busque una cadena en un archivo de texto; la cadena la determina el usuario.

Cuestionario de autoevaluación

1. ¿Qué es un archivo?
2. ¿Qué es un archivo de texto?
3. ¿Qué es un archivo binario?
4. ¿Qué es un flujo?
5. ¿Qué es un buffer?
6. ¿Cuál es la utilidad de la palabra **FILE**?
7. ¿Qué relación existe entre los apuntadores y los archivos?
8. ¿Cómo se puede crear un archivo binario?
9. ¿Cómo se puede leer un archivo binario?
10. ¿Cómo se puede actualizar un archivo binario?

Examen de autoevaluación

1. Función para leer un carácter:
   a. FILE
   b. remove
   c. printf
   d. getc
2. Función para leer una cadena de un archivo:
   a. EOF
   b. fgets
   c. fprintf
   d. gets

3. Función para abrir un flujo:
    a. FILE
    b. open
    c. fscanf
    d. fopen
4. La estructura de tipo FILE se encuentra definida en:
    a. stdio.h
    b. string.h
    c. stdlib.h
    d. conio.h
5. Función que salta al byte especificado en un flujo:
    a. feof
    b. getc
    c. fscanf
    d. fseek
6. Función para escribir en un archivo:
    a. fscanf
    b. fgetc
    c. fputs
    d. fgets
7. Función para cerrar un archivo:
    a. feof
    b. fclose
    c. fopen
    d. EOF
8. Función para borrar un archivo:
    a. rm
    b. delete
    c. remove
    d. del

9. Para actualizar un archivo sin eliminar su contenido se usa la letra:
    a. r
    b. b
    c. a
    d. w
10. El siguiente permiso permite crear un archivo para leer y escribir:
    a. r
    b. w
    c. w+
    d. b

Bibliografía Básica

BATALLER, Jordi y Rafael Magdalena, *Programación en C,* España, coedición Alfa omega-Universidad Politécnica de Valencia, 2004, 432 pp.

CEBALLOS, Francisco Javier, *JAVA 2, curso de programación,* México, Alfa omega-RaMa, 2ª Edición, 2004, 816 pp.

CEBALLOS, Francisco Javier, *El lenguaje de programación C#,* México, Alfa omega-RaMa, 2004, 320 pp.

DEITEL, H.M., DEITEL, P.J. "*Como programar en C/C++",* México: Prentice Hall, 2ª Edición, 1995, 927 pp.

FARRET, *Introducción a la programación. Lógica y diseño*, 4ª. Ed., México, Alfa omega, 2001.

HERNÁNDEZ, Roberto, *Estructuras de datos y algoritmos,* México, Prentice Hall, 2000, 296 pp.

JOYANES Aguilar Luis, *Programación en C++, algoritmos, estructuras de datos y objetos,* México, Mc. Graw-Hill, 2000.

JOYANES Aguilar Luis, *Estructuras de datos, algoritmos, abstracción y objetos,* México, Mc. Graw-Hill, 1998, 857 pp.

JOYANES Aguilar Luis, *Fundamentos de programación, libro de problemas,* México, Mc. Graw-Hill, 1997

LANGSSAM, Yedidyah, *Estructuras de datos con C y C++*, 2ª. Ed., México, Prentice Hall, 1997, 692 pp.

LEVINE Gutiérrez, Guillermo, *Computación y programación moderna, perspectiva integral de la informática*, México, Addison Wesley, 2001, 552 pp.

PEÑALOSA, Ernesto, *Fundamentos de programación C/C++, 4ª. Ed.,* México, coedición Alfa omega-RaMa, 2004, 572 pp.

RODRÍGUEZ, Carlos Gregorio, *Ejercicios de programación creativos y recreativos en C++,* México, Thomson, 2003.

Bibliografía complementaria

GARCÍA, Luis, Juan Cuadrado, Antonio De Amescua y Manuel Velasco, *Construcción lógica de programas, Teoría y problemas resueltos,* México, coedición Alfa omega-RaMa, 2004, 316 pp.

JOYANES, Luis. *"Fundamentos de Programación",* España: Pearson Prentice Hall, 3ª Edición, 2003, 1004 pp.

LÓPEZ, Leobardo, *Programación estructurada en turbo pascal 7,* México, Alfa omega, 2004, 912 pp.

LÓPEZ, Leobardo, *Programación estructurada, un enfoque algorítmico,* 2ª. Ed., México, Alfa omega, 2004, 664 pp.

SEDGEWICK, Robert, *Algoritmos en C++,* México, Adisson-Wesley Iberoamericana, 1995, 800 pp.

WEISS, Mark Allen, *Estructuras de datos en JAVA,* México, Addison Wesley, 2000, 740 pp.

Páginas Web

www.wikipedia.org
www.lawebdelprogramador.com
www.monografias.com

RESPUESTAS A LOS EXÁMENES DE AUTOEVALUACIÓN

INTRODUCCION A LA PROGRAMACIÓN

TEMA 1	TEMA 2	TEMA 3	TEMA 4	TEMA5	TEMA 6	TEMA 7
1. d	1. d	1. a	1. b	1. b	1. a	1. d
2. b	2. c	2. a	2. b	2. a	2. b	2. b
3. c	3. a	3. b	3. b	3. b	3. c	3. d
4. b	4. c	4. a	4. b	4. a	4. c	4. a
5. d	5. a	5. a	5. a	5. a	5. a	5. d
6. c	6. c	6. a	6. b	6. d	6. b	6. c
7. a	7. d	7. c	7. a	7. d	7. a	7. b
8. b	8. a	8. c	8. a	8. c	8. a	8. c
9. b	9. d	9. c	9. a	9. d	9. b	9. c
10. c	10. b	10. b	10. a	10. a	10. d	10. c